Module 7 Memory Management

Silberschatz: Chapter 8

In this module we will see that, to optimize the use of memory, the programs are scattered in memory according to different methods:
Pagination, segmentation

Memory management: objectives

- **Main memory usage optimization = RAM**
- **The greatest possible number of active processes must be kept there, in order to optimize the operation of the system in multiprogramming**
  - keep the system as busy as possible, especially the CPU
  - adapt to user memory needs
    - dynamic allocation as needed

Memory management: concepts in this chapter

- **Physical address and logical address**
  - physical memory and logical memory
- **Contiguous allowance**
  - partitions
- **Segmentation**
- **Paging**
- **Combined segmentation and pagination**

Application of these concepts

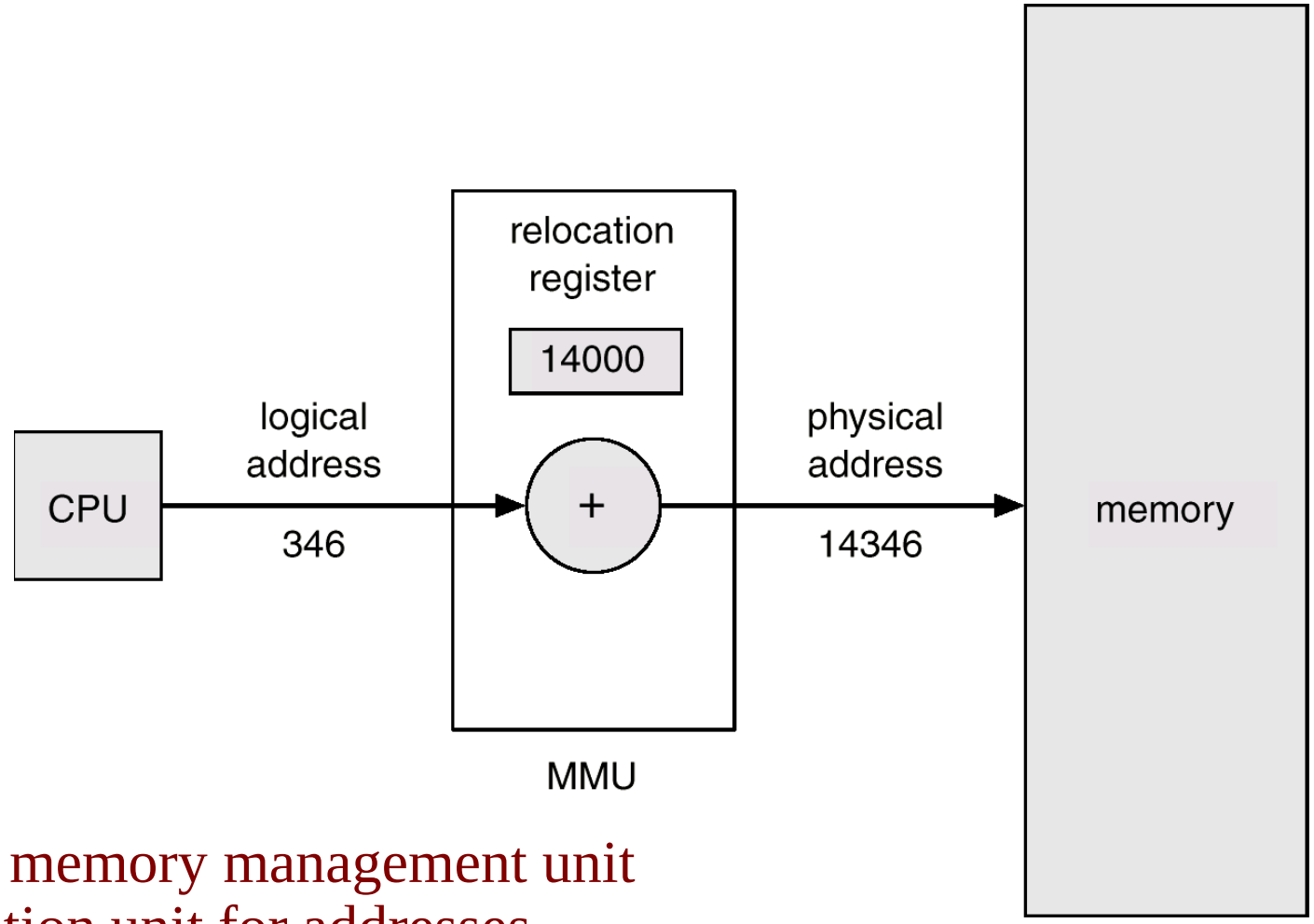
- Not all of the concepts in this chapter are actually used as is in today's main memory management.
- However, many are found in the field of auxiliary memory management, especially disks

Memory / Physical and logical addresses

- **Physical memory:**
  - the main memory RAM of the machine
- **Physical addresses: the addresses of this memory**
- **Logic memory: the address space of a program**
- **Logical addresses: the addresses in this space**

- **These concepts must be separated because normally, the programs are loaded from time to time in different memory positions.**
  - So physical address ≠ logical address

Logical address translation → physical addr.



MMU: memory management unit
translation unit for addresses
(memory management unit)

Definition of logical addresses

- a logical address is an address at a location in a program
  - compared to the program itself only
  - independent of the program position in physical memory

User view

- **Normally we have several types of addressing eg.**
  - the programmer's addresses (symbolic names) are translated at compile time into logical addresses
  - these addresses are translated into physical addresses after loading the program into memory by the address translation unit (MMU)

- **Given the wide variety of hardware and software, it is impossible to give more precise definitions.**

Binding logical and physical addresses (instructions and data)

- **The binding from logical addresses to physical addresses can be performed at different times:**
  - Compilation: when the physical address is known at the time of compilation (rare)
    - e.g. parts of the OS
  - Loading: when the physical address where the program is loaded is known, the logical addresses can be translated (rare today)
  - Runtime: normally, physical addresses are not known until runtime
    - e.g. dynamic allocation

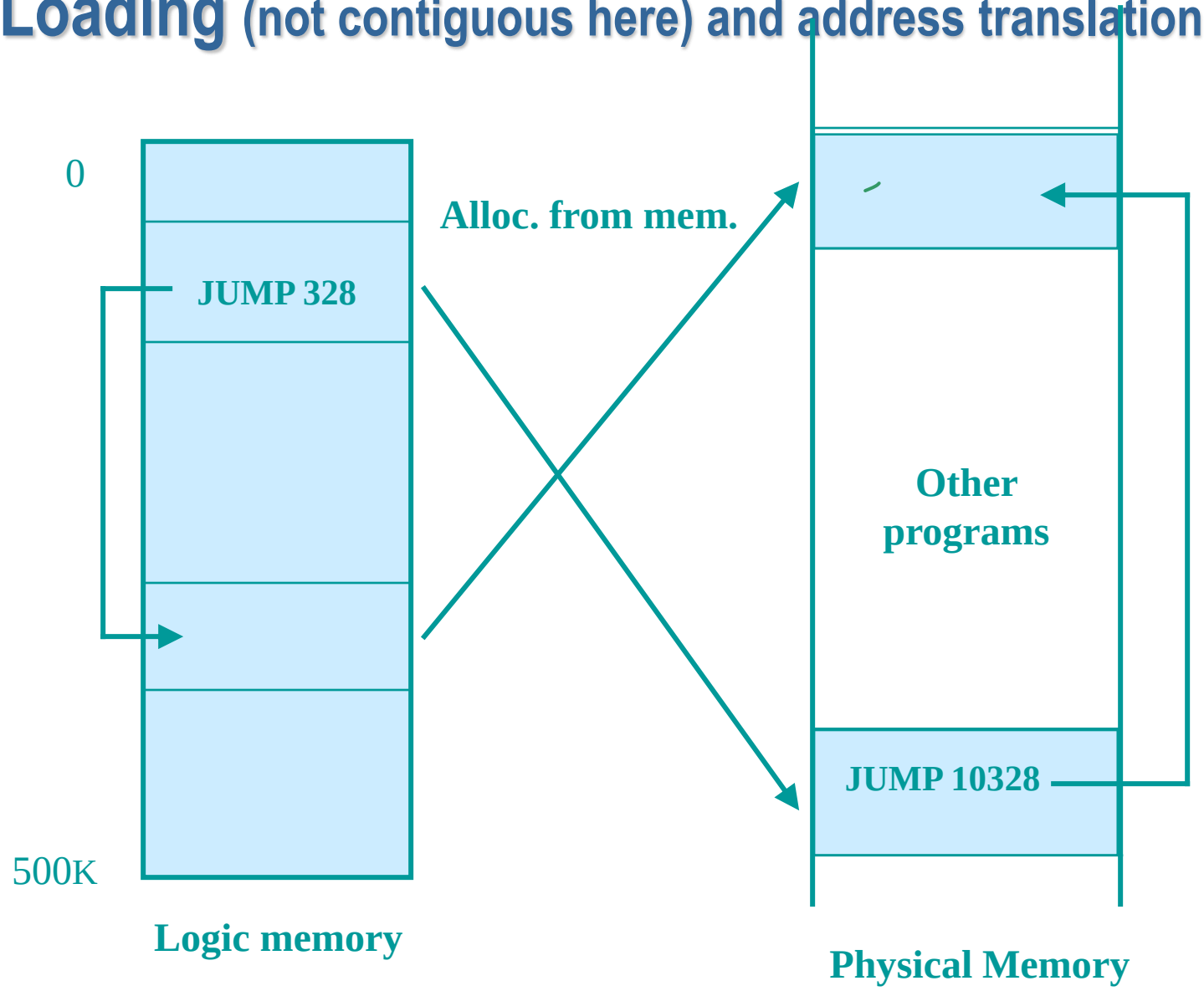
Two basic concepts

- **Loading. The program, or part of it, is loaded into physical memory, ready to run.**
  - static, dynamic
- **Linking of the different parts of a program to make it an executable entity.**
  - references between different modules must be translated
  - static (before execution)
  - dynamic (on request during execution)
    - Parts in the program = modules = segments = subprograms = objects, etc.

Loading aspects

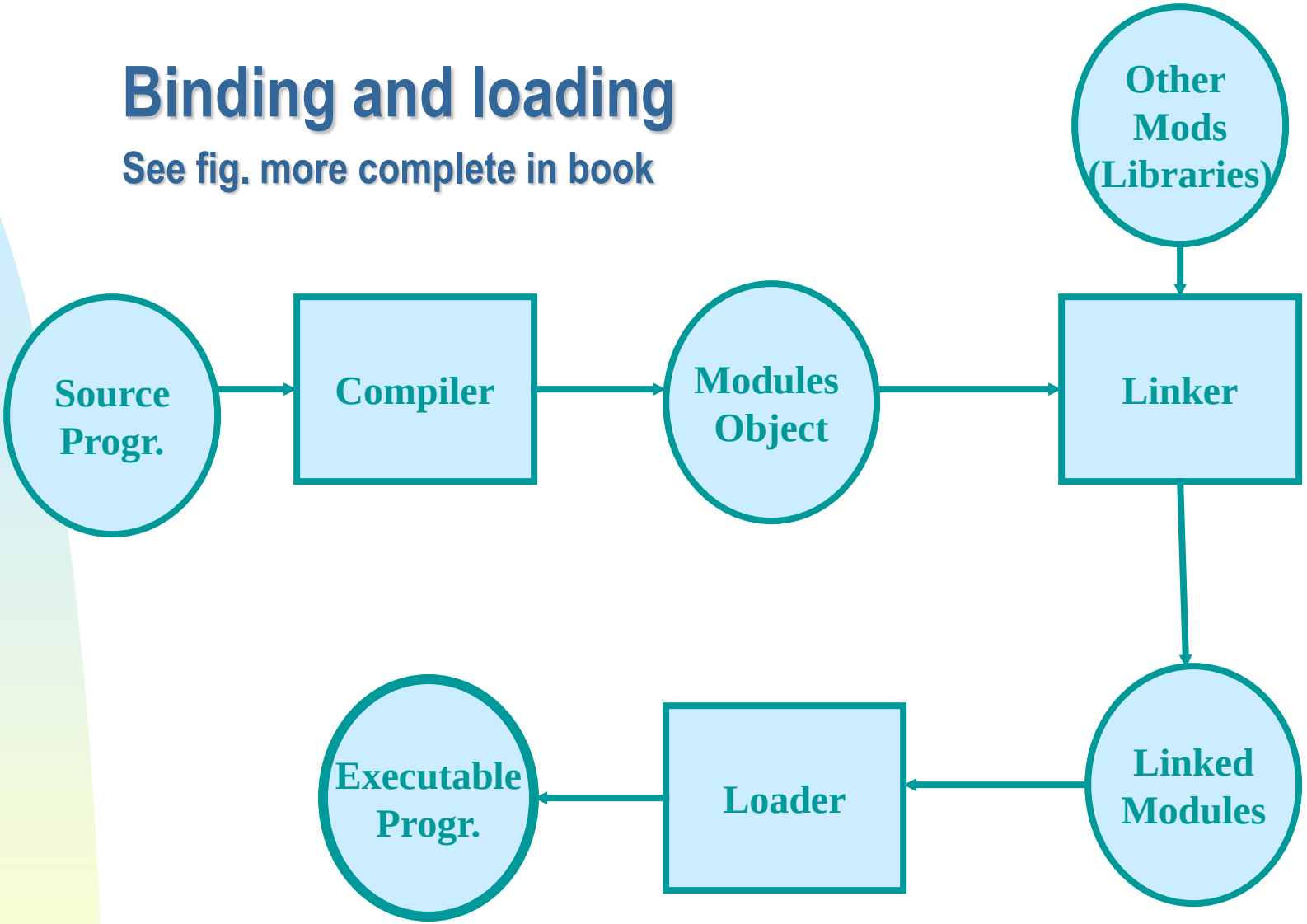
- **Find free memory for a loaded module: contiguous or not**
- **Translate the addresses of the program and make the connections in relation to the addresses where the module is loaded**

Loading (not contiguous here) and address translation



Binding and loading

See fig. more complete in book



NB: we assume that all the modules are known at the beginning
Often this is not the case → dynamic loading

Loading and dynamic binding

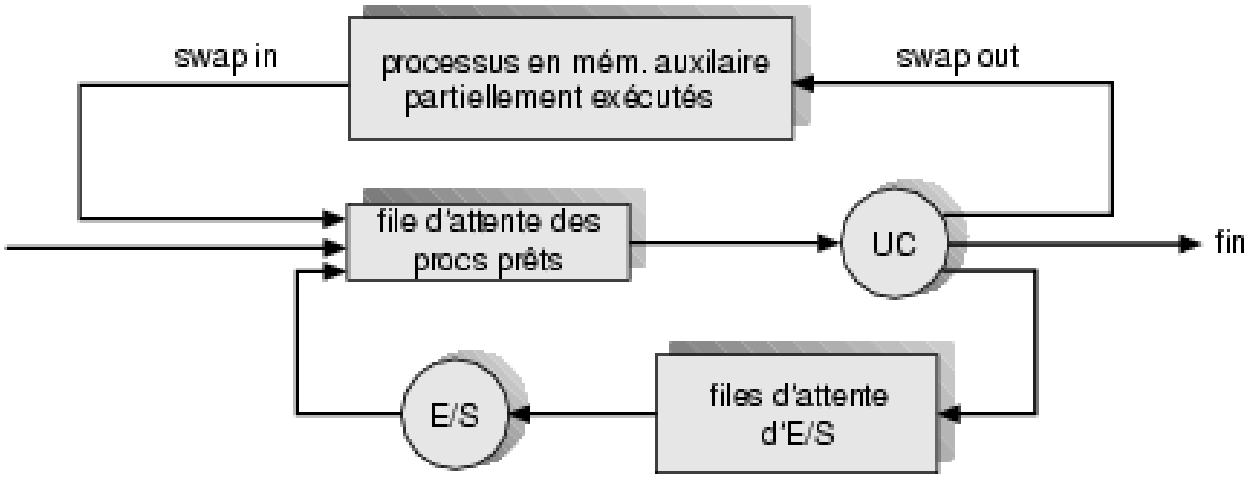
- **A running process may need different program modules at different times**
- **Static loading can therefore be inefficient**
- **It is better to load modules on demand (i.e. dynamically)**
  - dll, dynamically linked libraries
- **In a program which may need to load modules dynamically, at the beginning these are represented by *stubs* which indicate how to get to the modules (e.g. where it is: disk, www, other ...)**
- **At its 1st execution. the stub causes the module to be loaded into memory and it's linked with the rest of the program**
  - dynamic binding
- **Successive invocations of the module must not pass through this, we will know the address in memory**

Logical address translation → physical

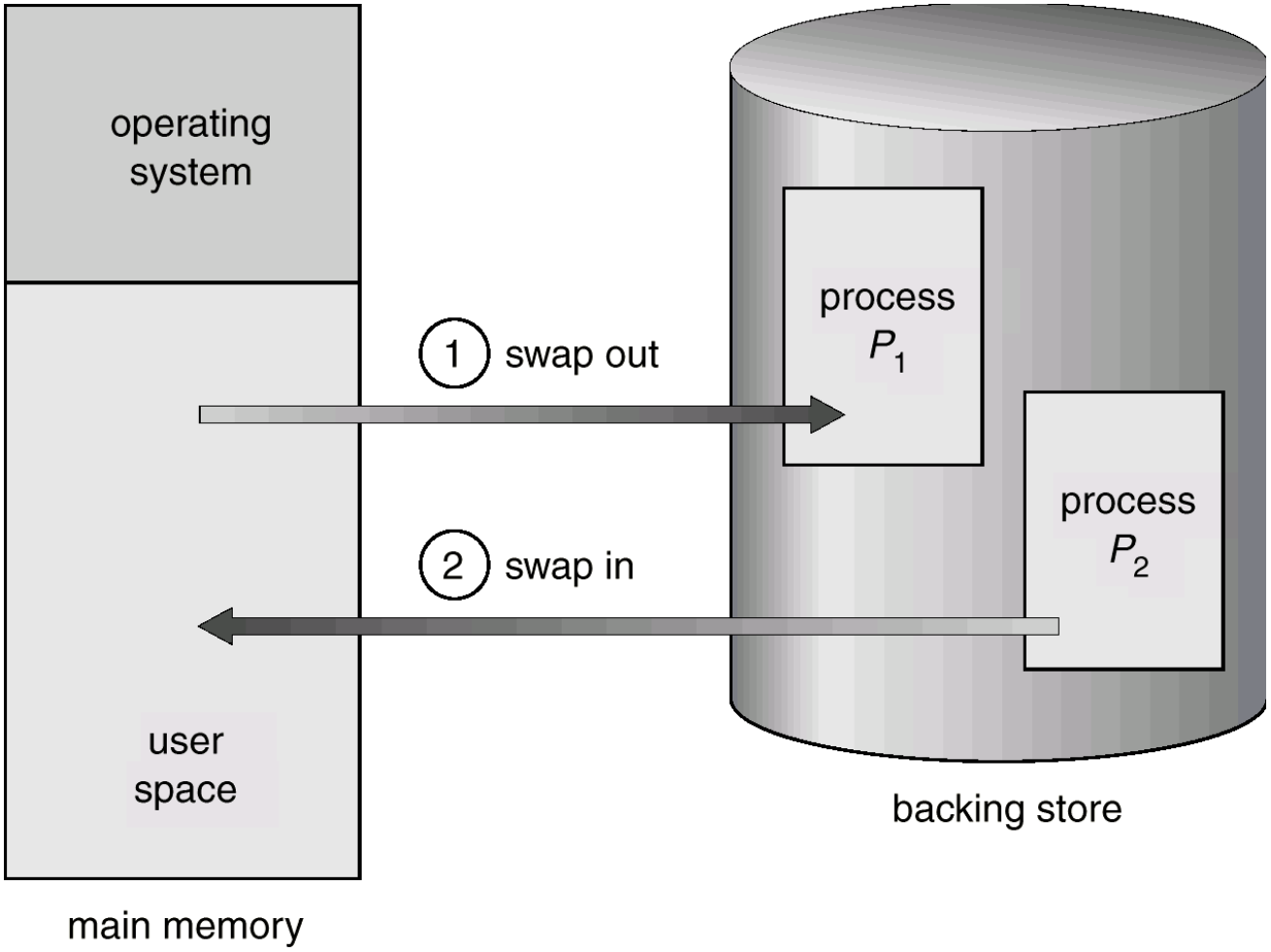
- **In early systems, a program was always read at the same memory addresses**
- **Multiprogramming and dynamic allocation created the need to read a program in different positions**
- **In the beginning, this was done by the loader which changed the addresses before launching the execution.**
- **Today this is done by the MMU as the progress. is executed**
- **This does not cause increased execution time, as the MMU acts in parallel with other CPU functions.**
  - Eg. the MMU can prepare the address of an instruction at the same time as the CPU executes the previous instruction

Program swapping

- **A program, or part of a program, can be temporarily removed from memory to allow the execution of other programs (chap. 4)**
  - it is put in secondary memory, normal. disk



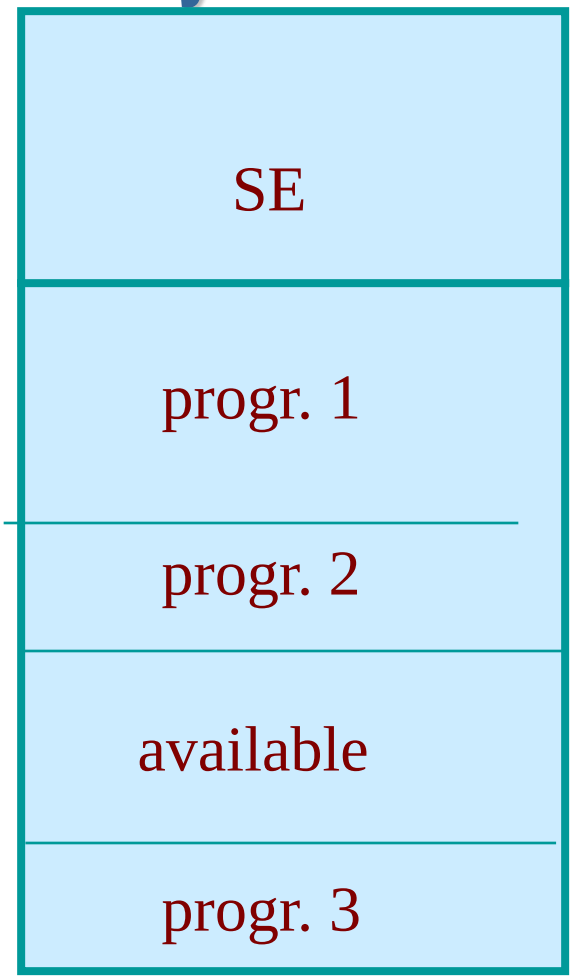
Program swapping



Contiguous memory allocation

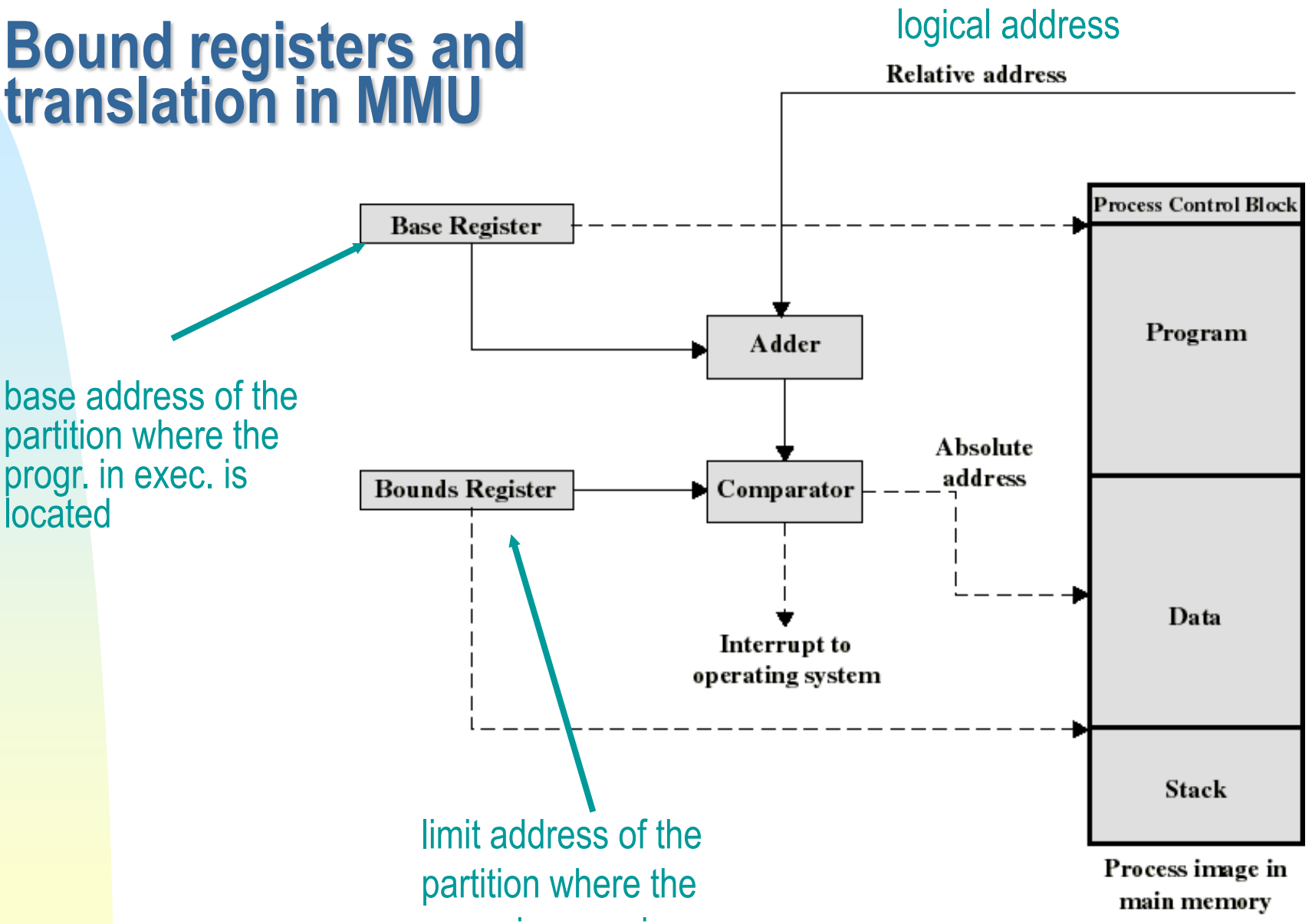
- **We have several programs to run**
- **We can load them into memory one after the other**
  - the place where a program is read is only known when loading
- **Material requirements: translation registers and bounds registers**

Contiguous memory allocation



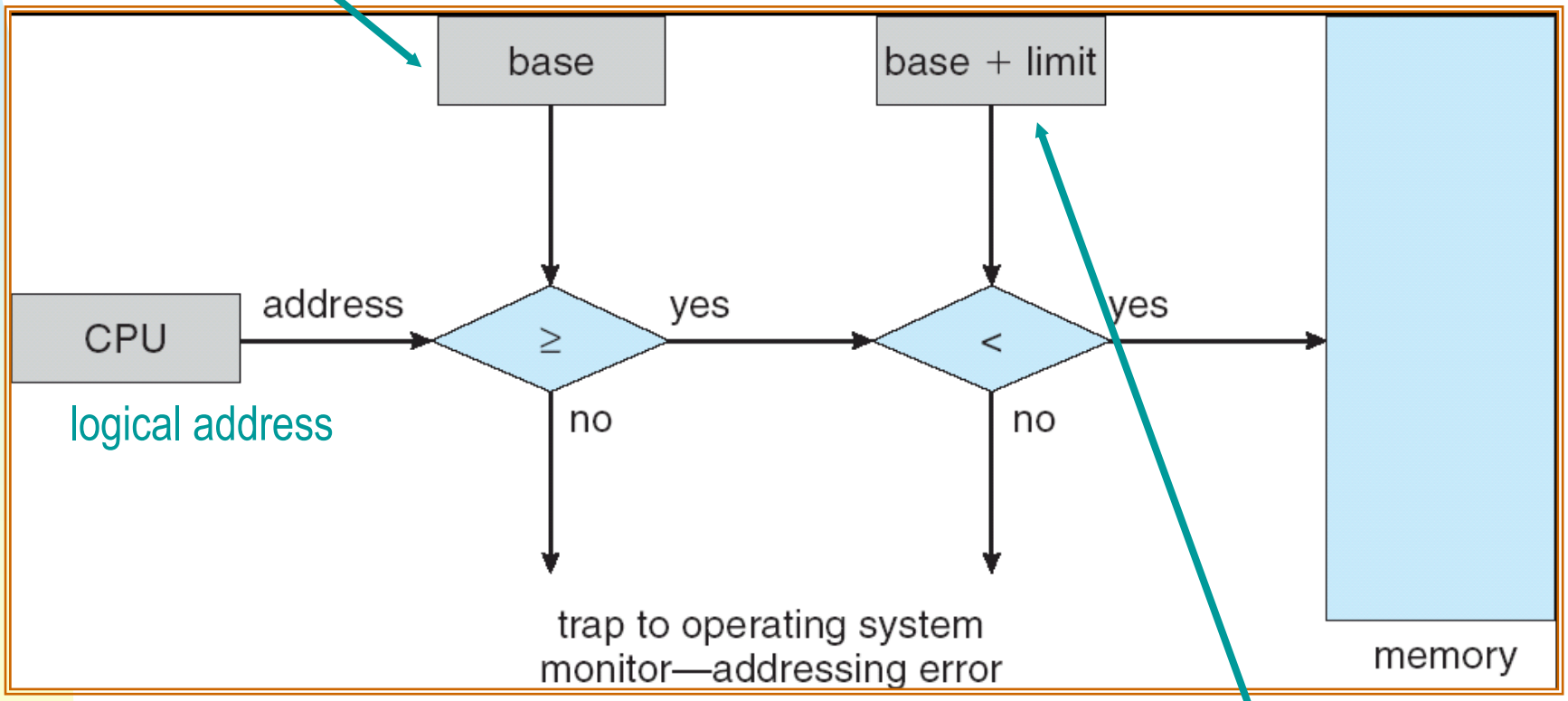
Here we have 4 partitions for programs - each is read in a single memory area

Bound registers and translation in MMU



Bound registers and translation in MMU

base address of the partition where the progr. in exec. is located



limit address of the partition where the progr. in exec. is located

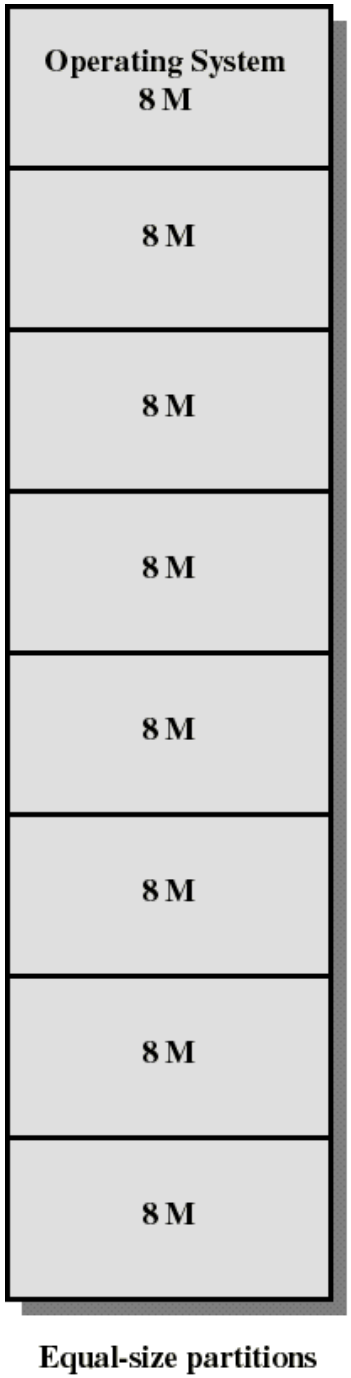
Fragmentation: unused memory

- **A major problem in contiguous allocation:**
  - There is enough space to run a program, but it is non-contiguously fragmented
    - external: unused space is Between partitions
    - internal: unused space is in the partitions

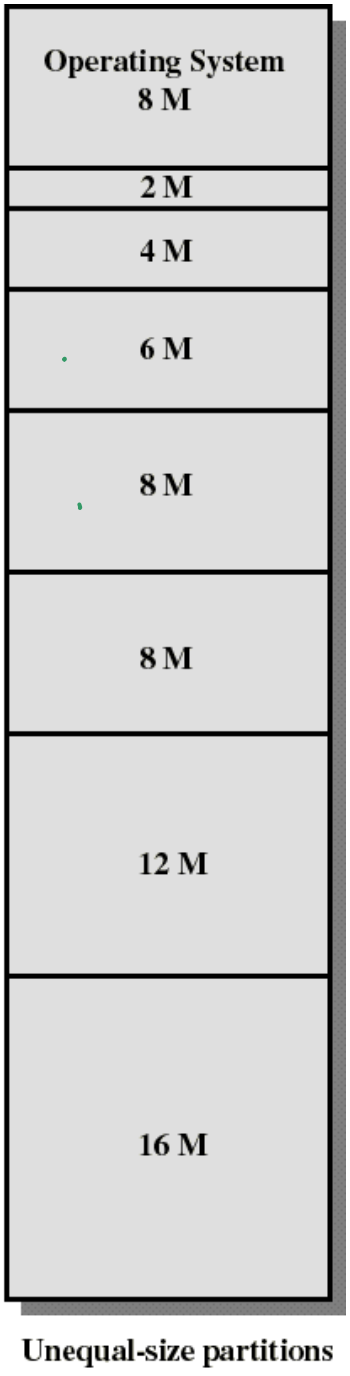
Fixed partitions

- Main memory subdivided into distinct regions: partitions
- Partitions are either the same size or unequal sizes
- Any progr. can be allocated to a partition that is large enough

(Stallings)



Equal-size partitions



Unequal-size partitions

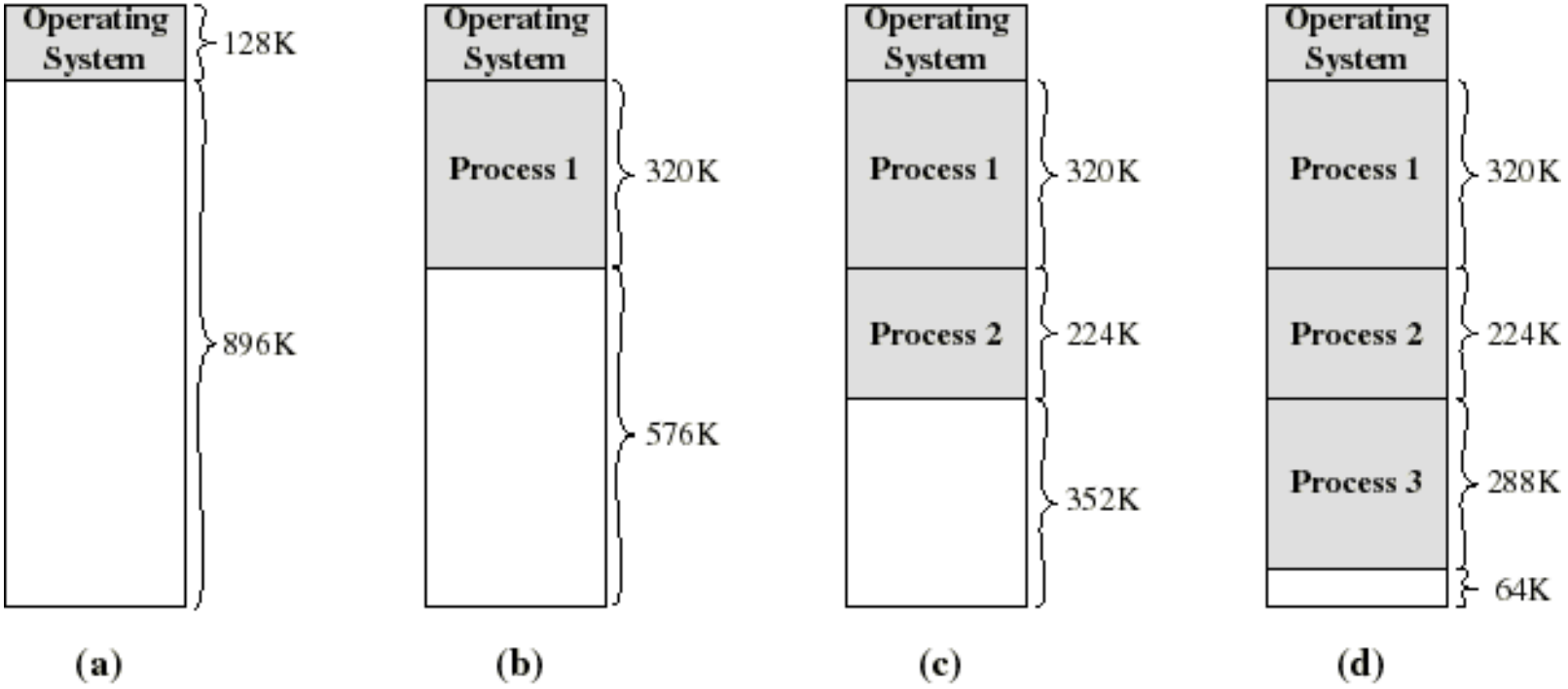
Fixed partitions

- Simple, but ...
- Inefficient memory usage: Any program, no matter how small, must occupy an entire partition. There is internal fragmentation.
- Partitions of unequal sizes alleviate these problems but they remain there ...

Dynamic partitions

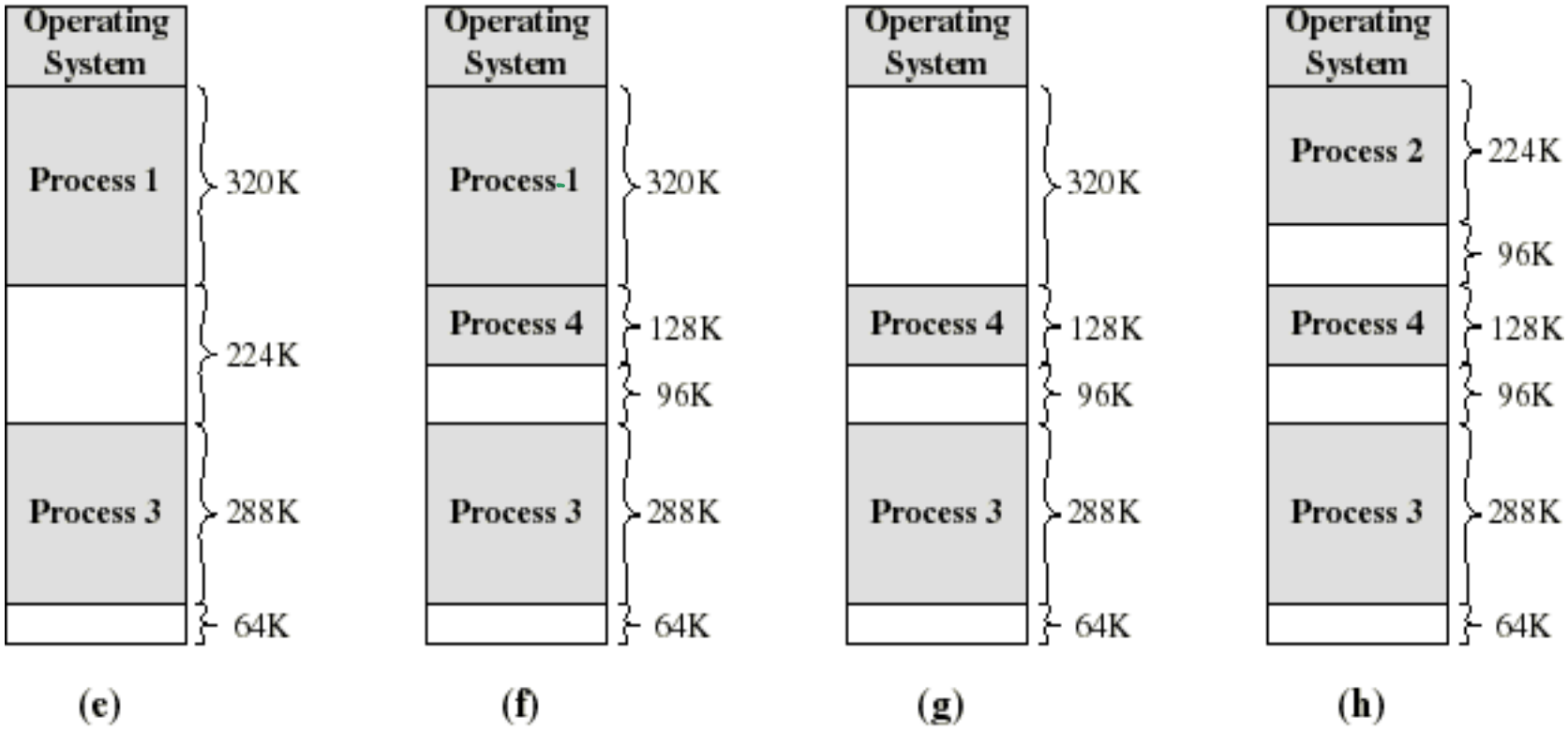
- Partitions in varying numbers and sizes
- Each process is allocated exactly the required memory size
- Probably unused holes will form in the memory: this is the external fragmentation

Dynamic partitions: example (Stallings)



- (d) There is 64K hole after loading 3 processes: not enough space for other process
- If all proc get stuck (e.g. waiting for an event), P2 can be swapped out and P4 = 128K can be loaded.

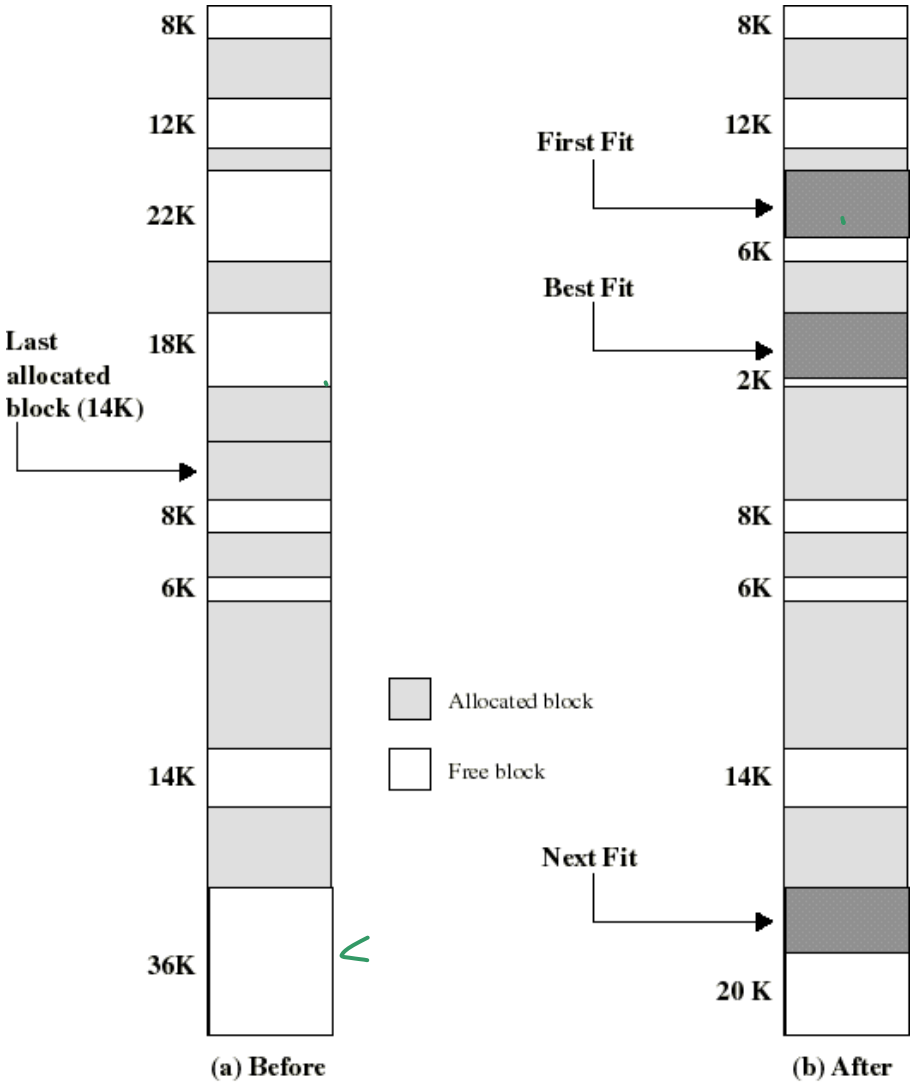
Dynamic partitions: example (Stallings)



- (e-f) Progr. 2 is suspended, Progr. 4 is loaded. A hole of 224-128 = 96K is created (*external fragmentation*)
- (g-h) P1 ends or it is suspended, P2 is put back in its place: producing another hole of 320-224 = 96K ...
- We have 3 small holes and probably unused. 96 + 96 + 64 = 256K external fragmentation
- COMPRESSION to make a single hole of 256K

Placement Algorithm in dynamic allocation

- to decide the location of the next process
- Goal: reduce compression usage (takes time...)
- Possible choices:
  - "Best-fit": choose the smallest hole
  - "First-fit": choose 1st hole from the beginning
  - "Next-fit": choose 1st hole from the last placement



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

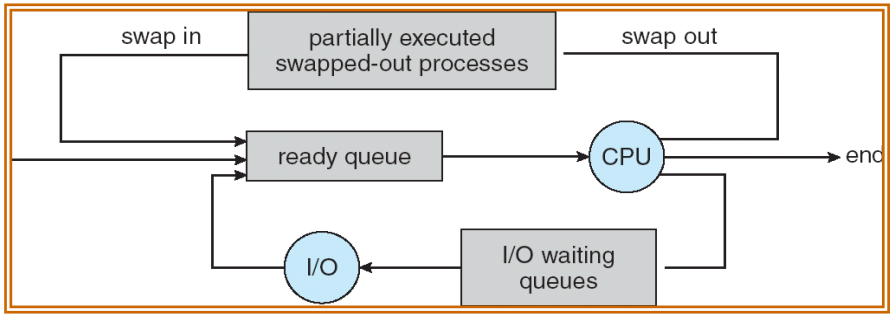
(Stallings)

Placement Algorithms: comments

- **What is the best?**
  - main criterion: decrease the probability of situations where a process cannot be served, even if there is enough memory ...
- **The simulation shows that it is not worth using the most complex algorithms ... so first fit**
- **“Best-fit”: look for the smallest possible block: the hole created is as small as possible**
  - memory will be filled with holes too small to hold a program
- **Next-fit ”: allocations will often be done at the end of memory**

Suspension (see chap 4)

- **When all the programs in memory are blocked, the OS can suspend one (swap / suspend)**
  - We transfer to the disk one of the blocked processes (thus putting it in suspended state) and replace it with a process ready to be executed
    - the latter process performs a transition from New or Suspended to Ready state



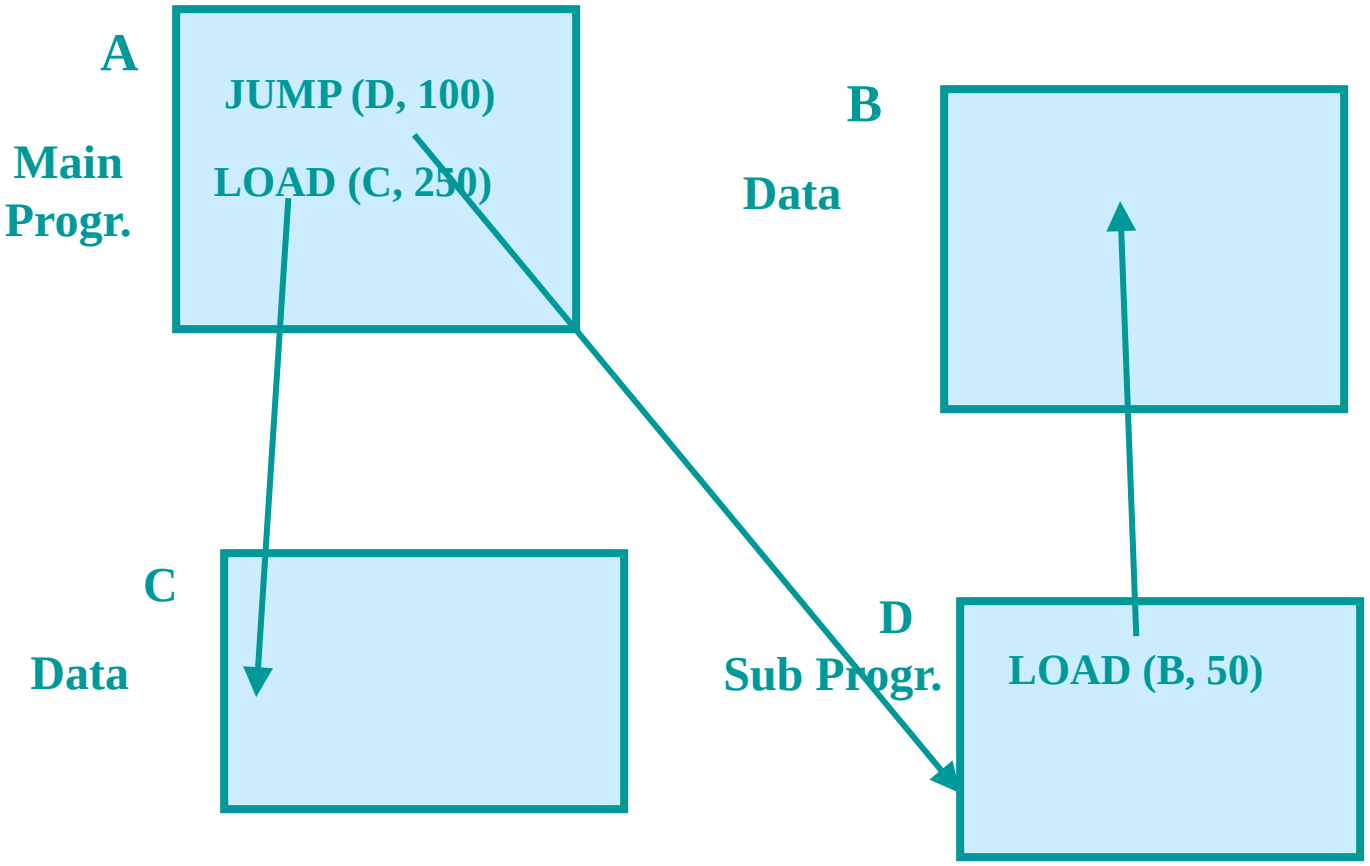
Compression (compaction)

- **A solution for external fragmentation**
- **The programs are moved in memory so as to reduce to 1 single large hole several small holes available**
- **Performed when a program requesting to be executed does not find a partition large enough, but its size is smaller than the existing external fragmentation**
- **Disadvantages:**
  - program transfer time
  - need to re-establish all links between addresses of different programs

Non-contiguous allowance

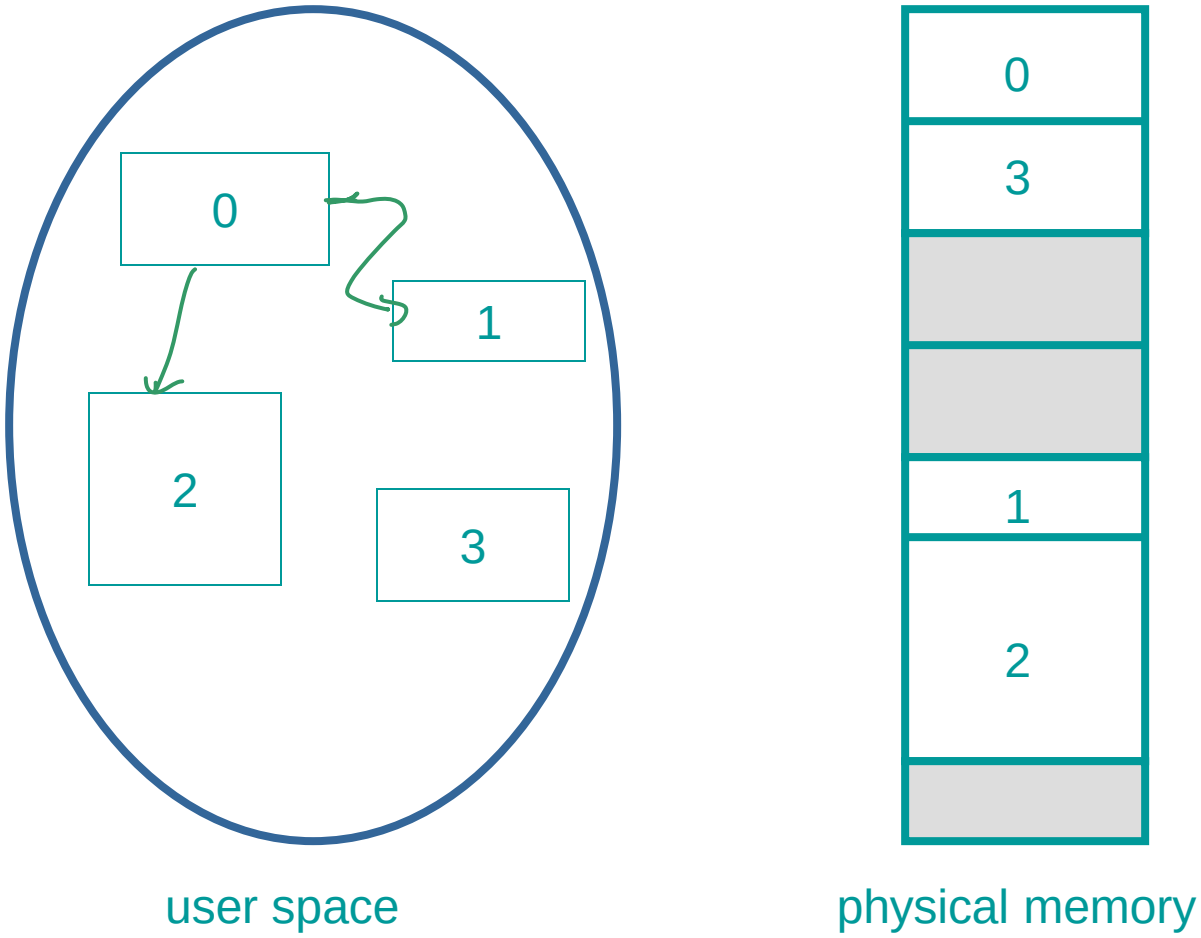
- **To reduce the need for compression, the next step is to use the non-contiguous allocation**
  - split a program into chunks and allow separate allocation of each chunk
  - the pieces are much smaller than the entire program and therefore allow more efficient use of memory
    - small holes can be used more easily
- **There are two basic techniques for doing this: pagination and segmentation.**
  - segmentation uses program parts that have a logical value (modules)
  - pagination uses arbitrary parts of the program (breaking up the program into pages of fixed length).
  - they can be combined
- **I find the segmentation to be more natural, so I start with this one**

Segments are logical parts of the program.



4 segments: A, B, C, D

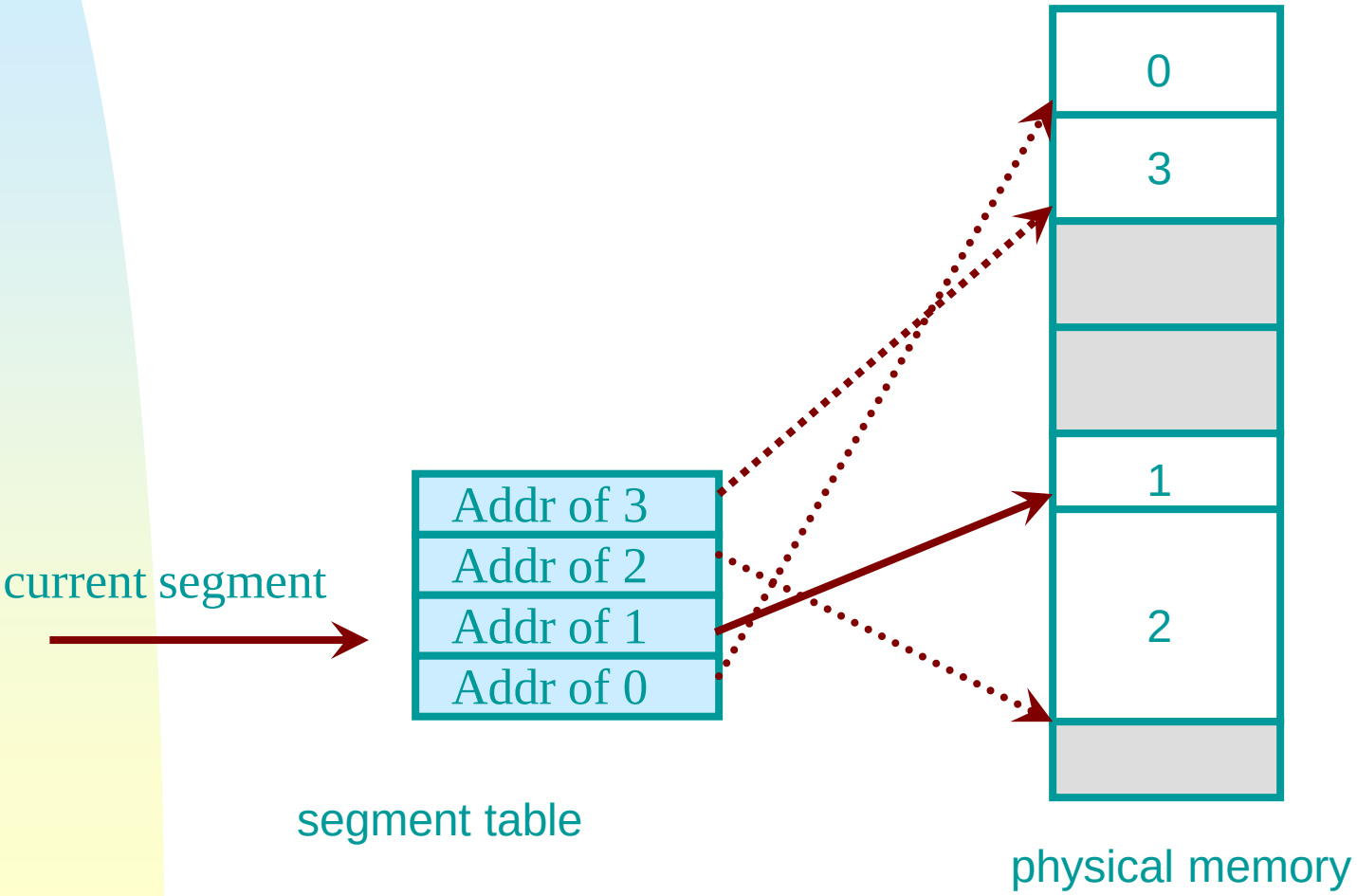
Segments as memory allocation units



Since segments are smaller than entire programs, this technique involves less fragmentation (which is external in this case)

Mechanism for segmentation

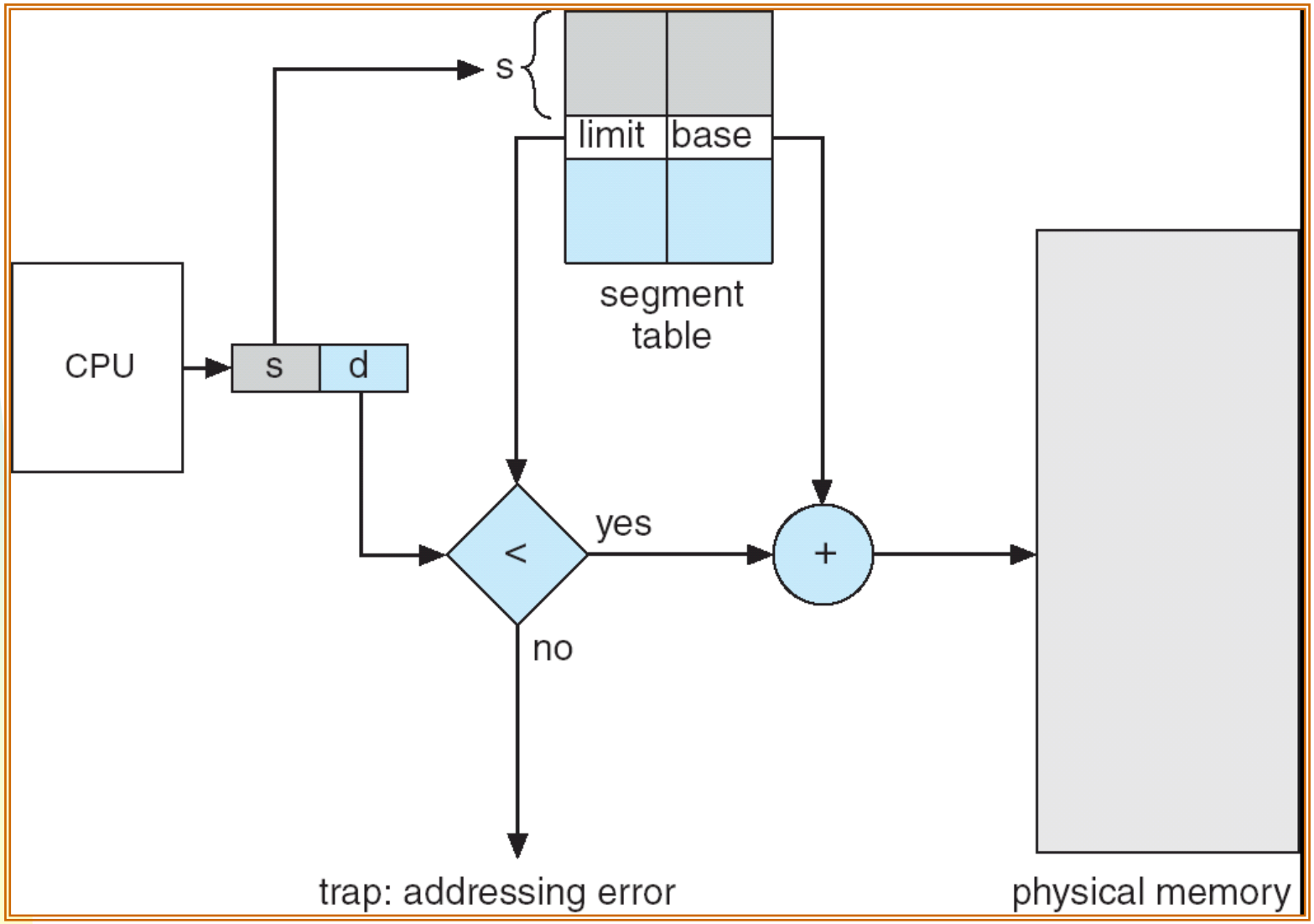
- A table contains the starting address of all segments in a process
- Each address in a segment is appended to the segment start address by the MMU



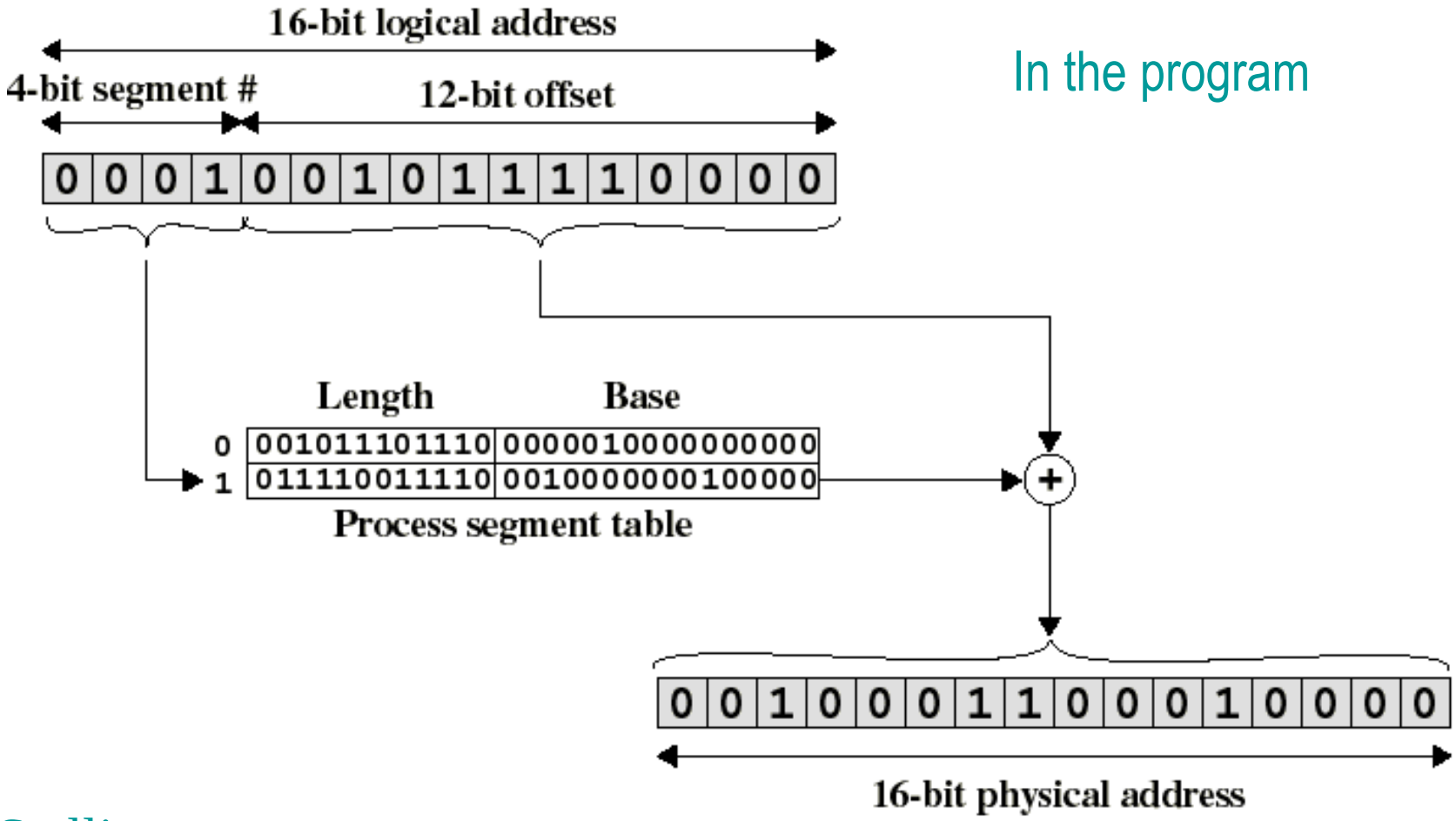
Details

- **The logical address consists of a pair:**
  - **<No of segm, offset>**
  - where offset is the address *in* the segment
- **the segment table contains: segment descriptors**
  - base address
  - segment length
  - Protection info, we'll see ...
- **In the process PCB there will be a pointer to the memory address of the segment table.**
- **There will also be the number of segments in the process**
- **At the time of context switching, this info will be loaded into the appropriate CPU registers.**

Address translation in segmentation



The mechanism in detail (implemented in the hardware)

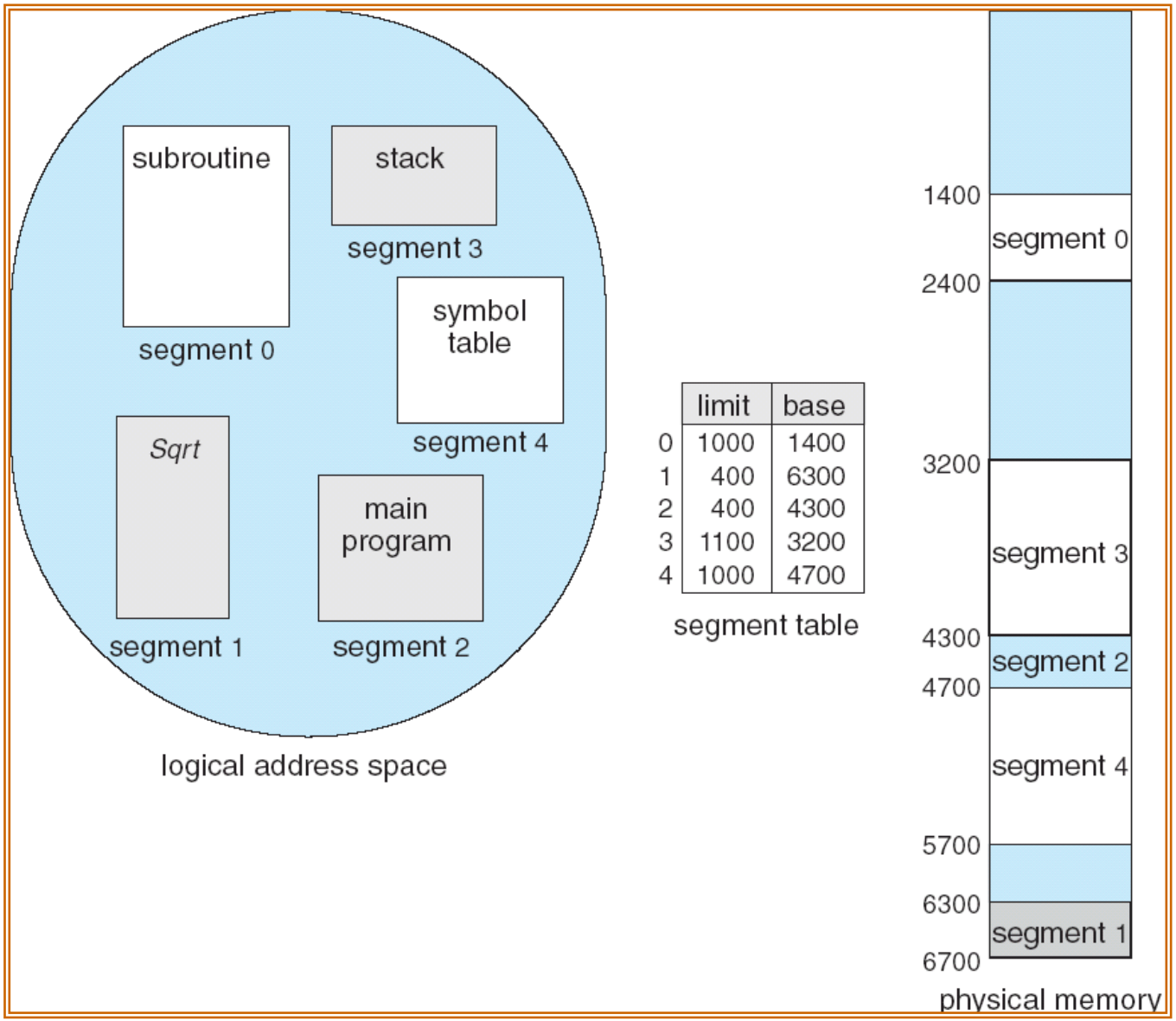


In the program

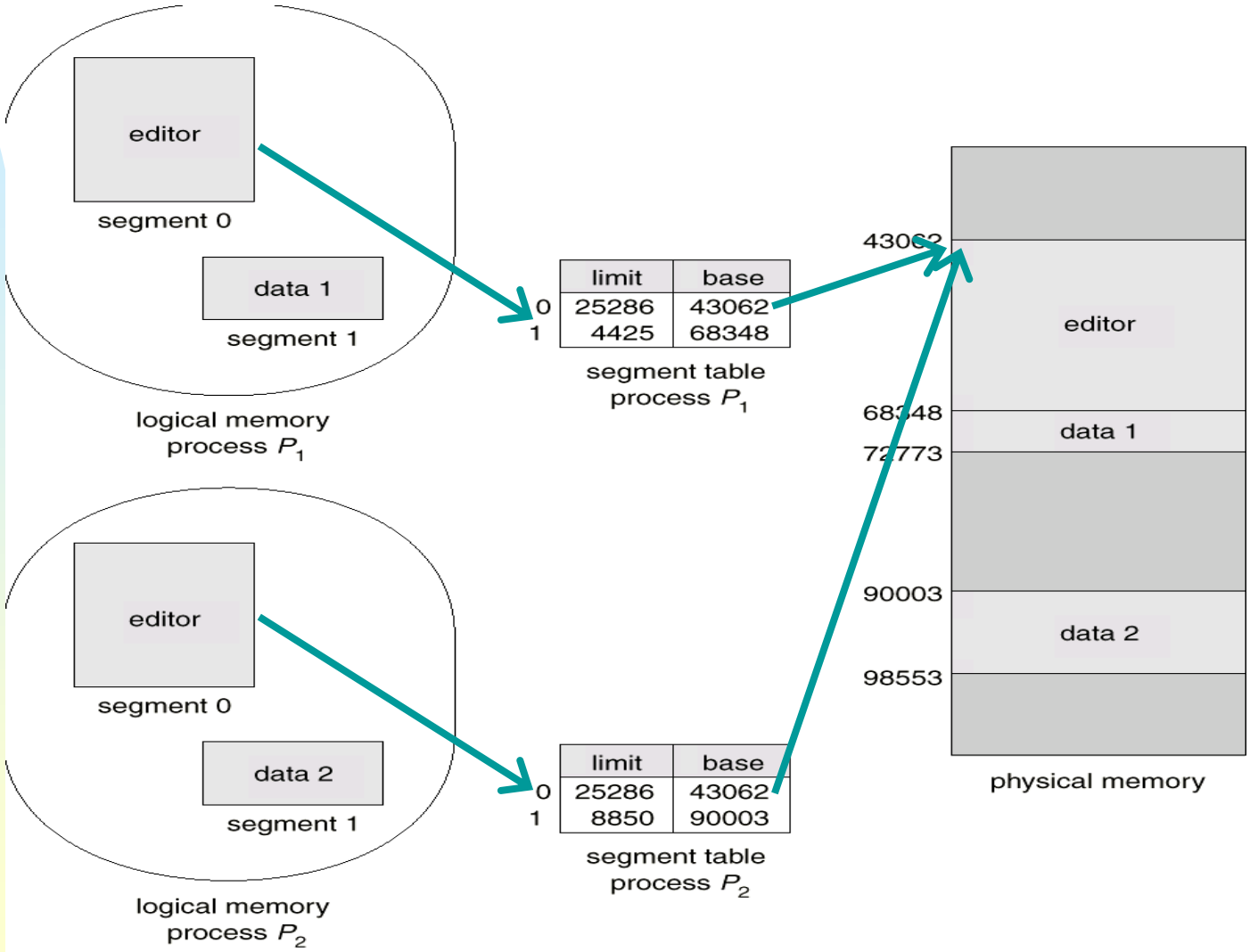
Stallings

Final address

Example of simple segmentation

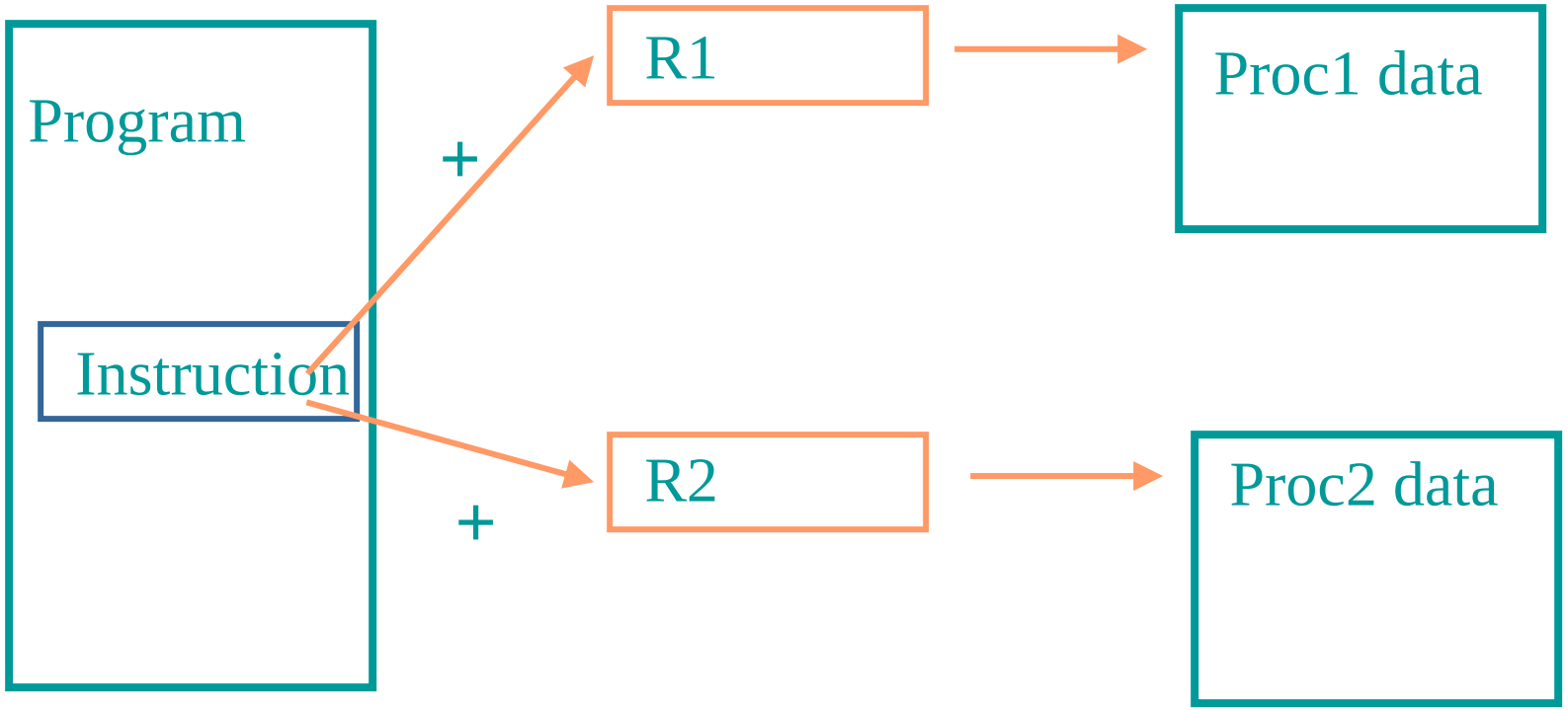


Sharing segments: segment 0 is shared



Eg: DLL used by several users

Mechanism for 2 processes that execute a single program on different data



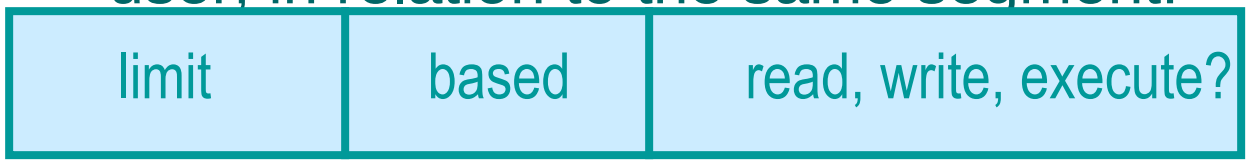
The same instruction, if executed

- by proc 1, its address is modified by the content of base reg 1
- by proc 2, its address is modified by the content of base reg 2

This works even if the instruction is executed by more. CPUs at the same time, if the registers are in different CPUs

Segmentation and protection

- **Each entry in the segment table can contain protection info:**
  - segment length
  - user privileges on the segment: read, write, execute
    - If, when calculating the address, it is found that the user does not have access rights→interruption
    - this information can therefore vary from user to user, in relation to the same segment!



Simple segmentation assessment

- **Advantages: The memory allocation unit is**
  - smaller than the whole program
  - a logical entity known by the programmer
  - segments can change places in memory
  - protecting and sharing segments is easy (in principle)
- **Disadvantage: the problem of dynamic partitions:**
  - External fragmentation is not eliminated:
    - holes in memory, compression?
- **Another solution is to try to simplify the mechanism by using memory allocation units of equal sizes.**
    - **PAGING**

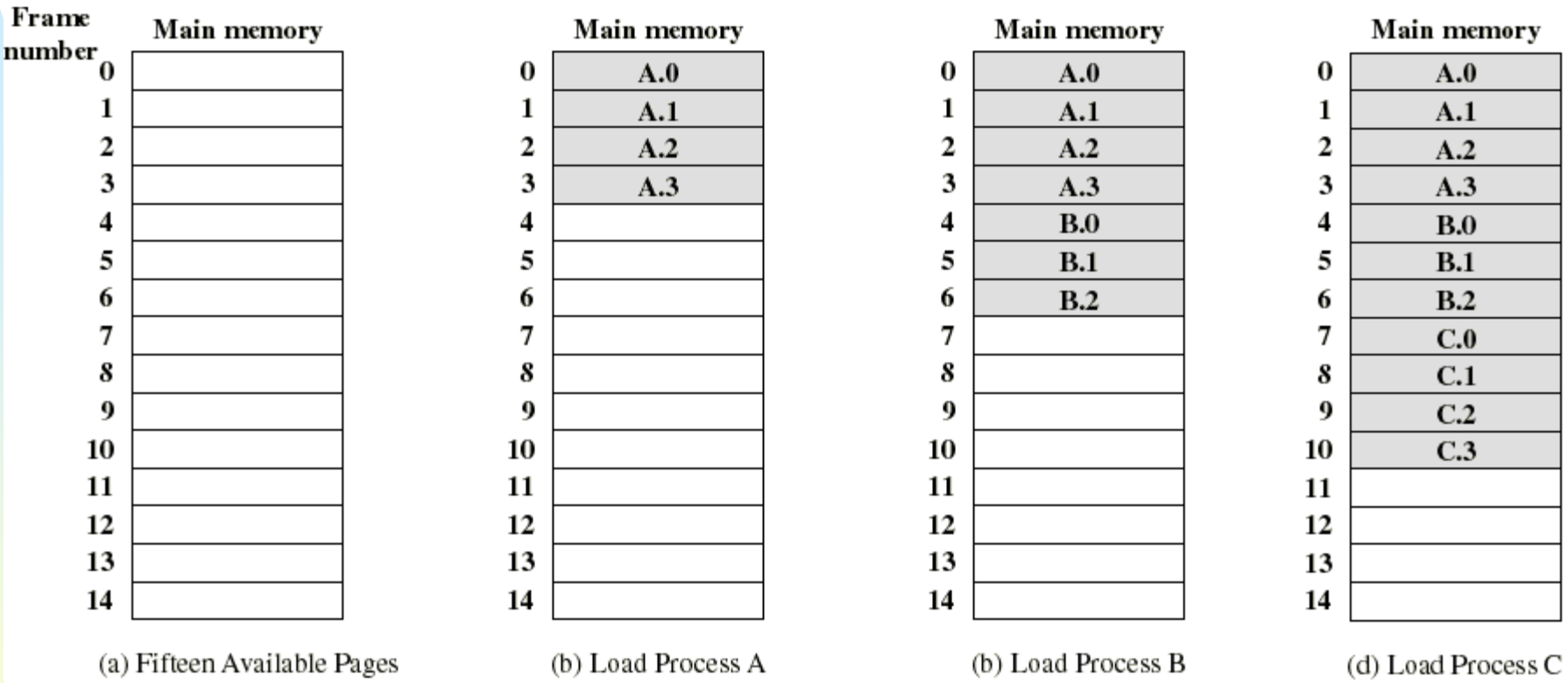
Segmentation versus pagination

- The pb with segmentation is that the memory allocation unit (the segment) is of variable length
- Paging uses fixed memory allocation units, therefore eliminating this pb

Simple pagination

- **The memory is partitioned into small pieces of the same size: the physical pages or 'frames'**
- **Each process is also partitioned into small pieces of the same size called pages (logical)**
- **Logical pages of a process can therefore be assigned to frames available anywhere in main memory.**
- **Consequences:**
  - a process can be scattered anywhere in physical memory.
  - fragmentation **external** is eliminated

Example of process loading

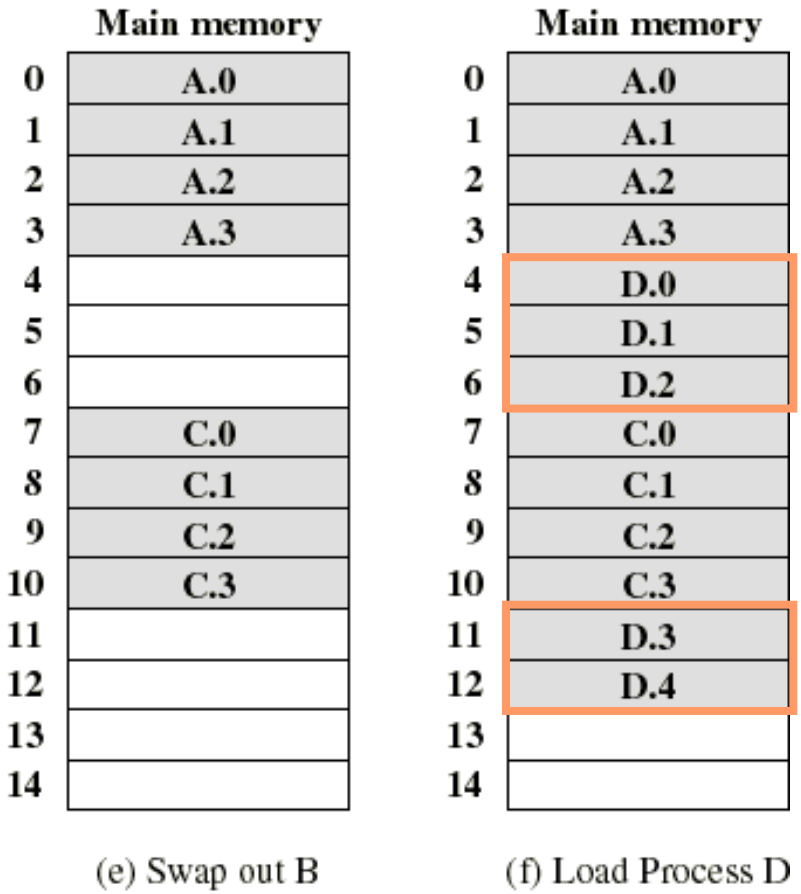


- **Suppose process B ends or is suspended**

Stallings

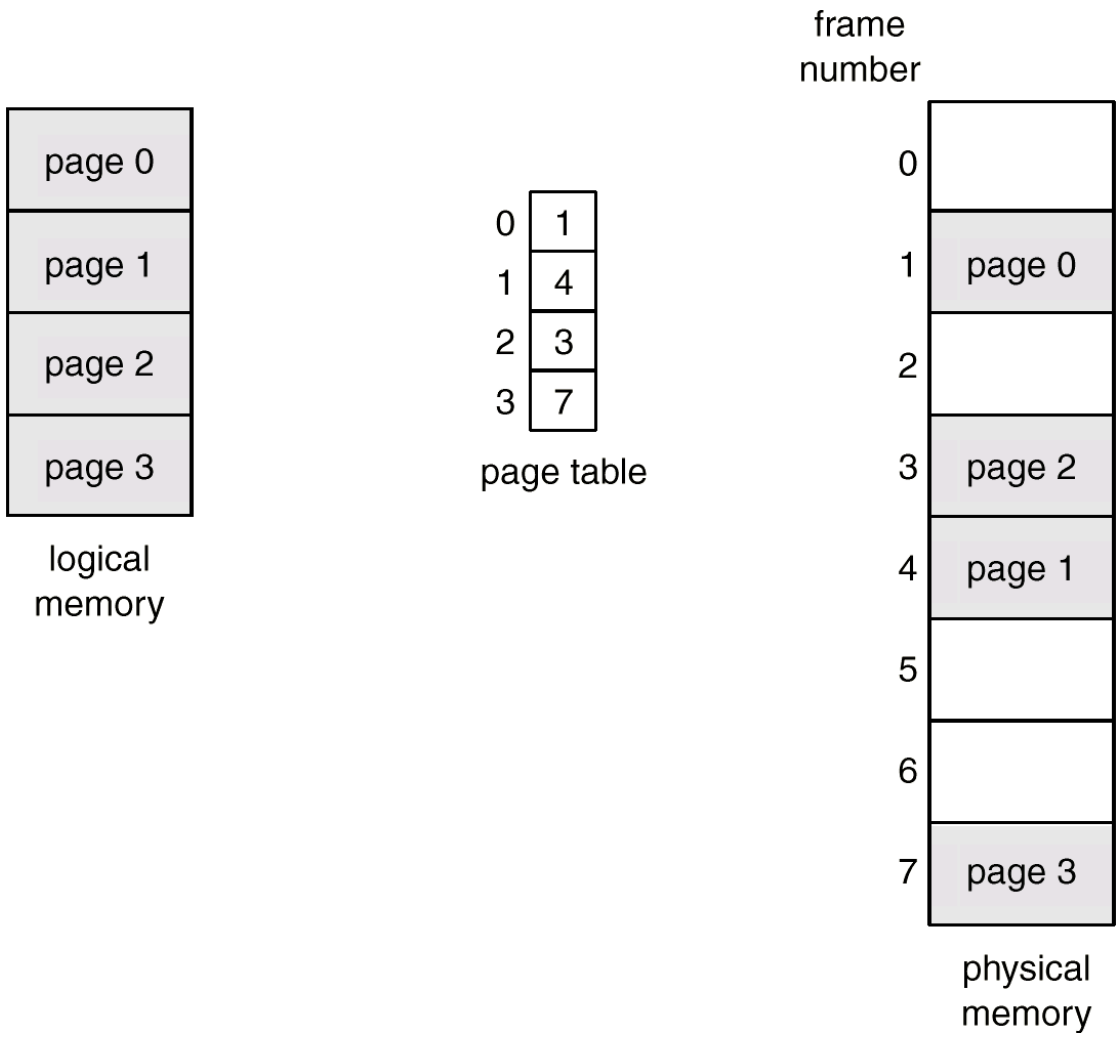
Example of process loading (Stallings)

- **We can now transfer a program D to memory which requires 5 frames**
  - although there are no 5 contiguous frames available
- **External fragmentation is limited to the case that the number of available pages is not sufficient to run a waiting program.**
- **Only the last page of a process can suffer from internal fragmentation (avg. 1/2 frame per proc)**



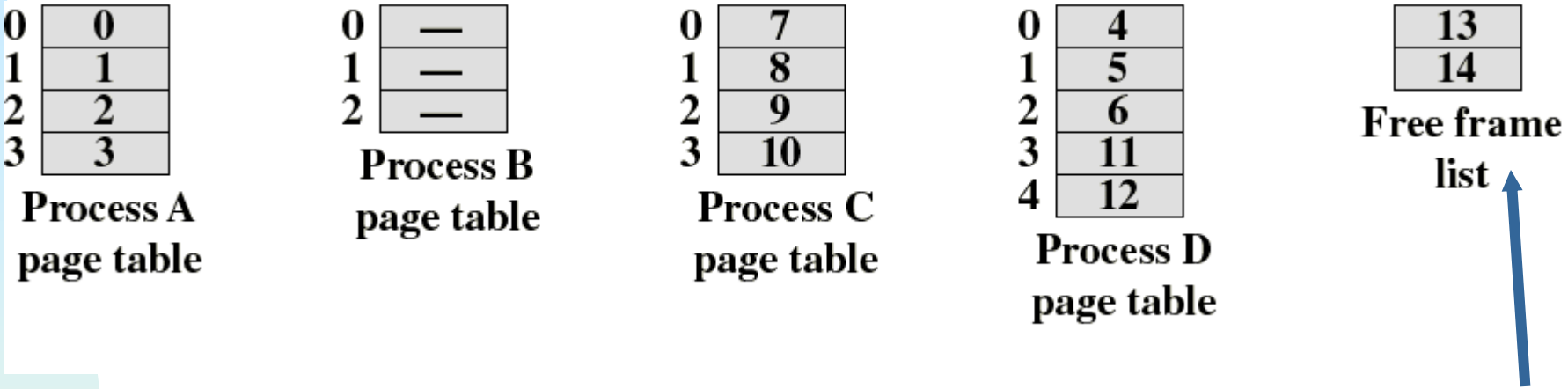
(e) Swap out B (f) Load Process D

Page tables


page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
frame number
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

Page tables

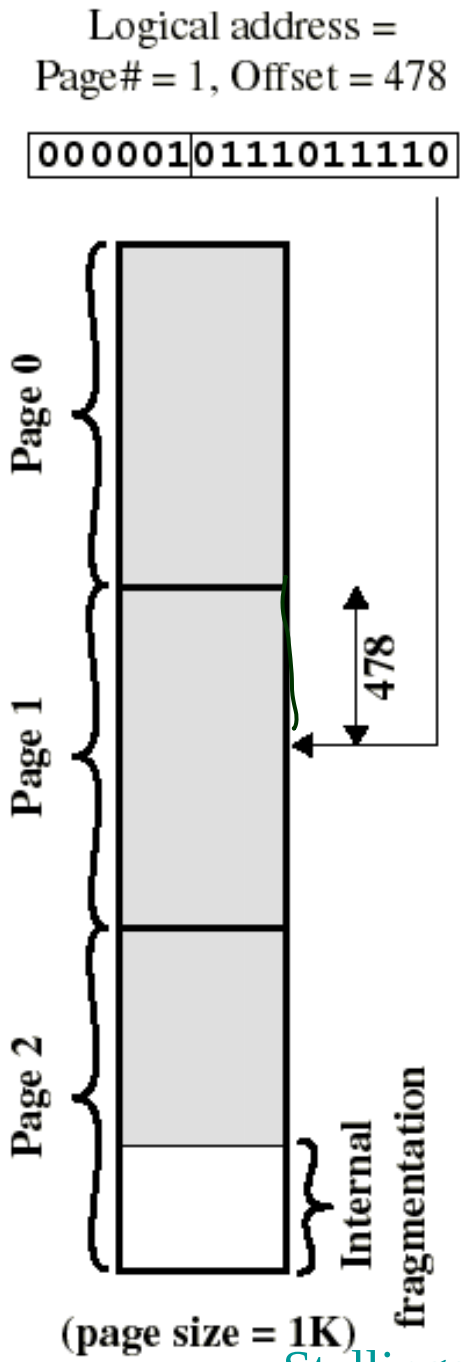
Stallings



- The OS must maintain a page table for each process
- Each entry in a page table contains the frame number where the corresponding page is physically located
- A page table is indexed by the page number in order to obtain the frame number
- A list of available frames is also maintained (free frame list)

Logical address (pagination)

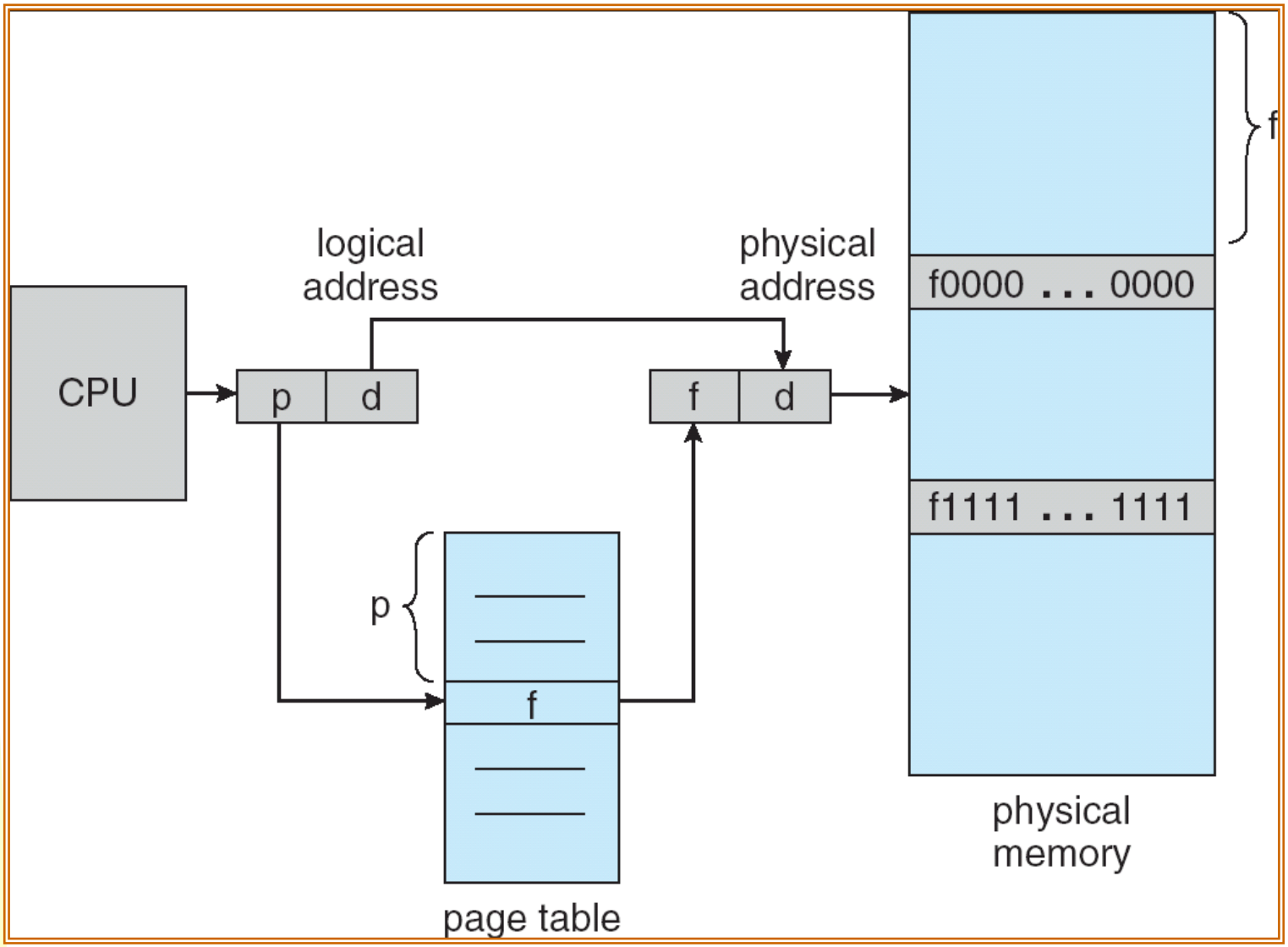
- The logical address is easily translated into a physical address because the page size is a power of 2
- Ex: if 16 bits are used for the addresses and the size of a page = 1K: we need 10 bits for the offset, thus leaving 6 bits for the page number
- The logical address (n, m) is translated to the physical address (k, m) using n as an index on the page table and replacing it with the address k found
  - m does not change



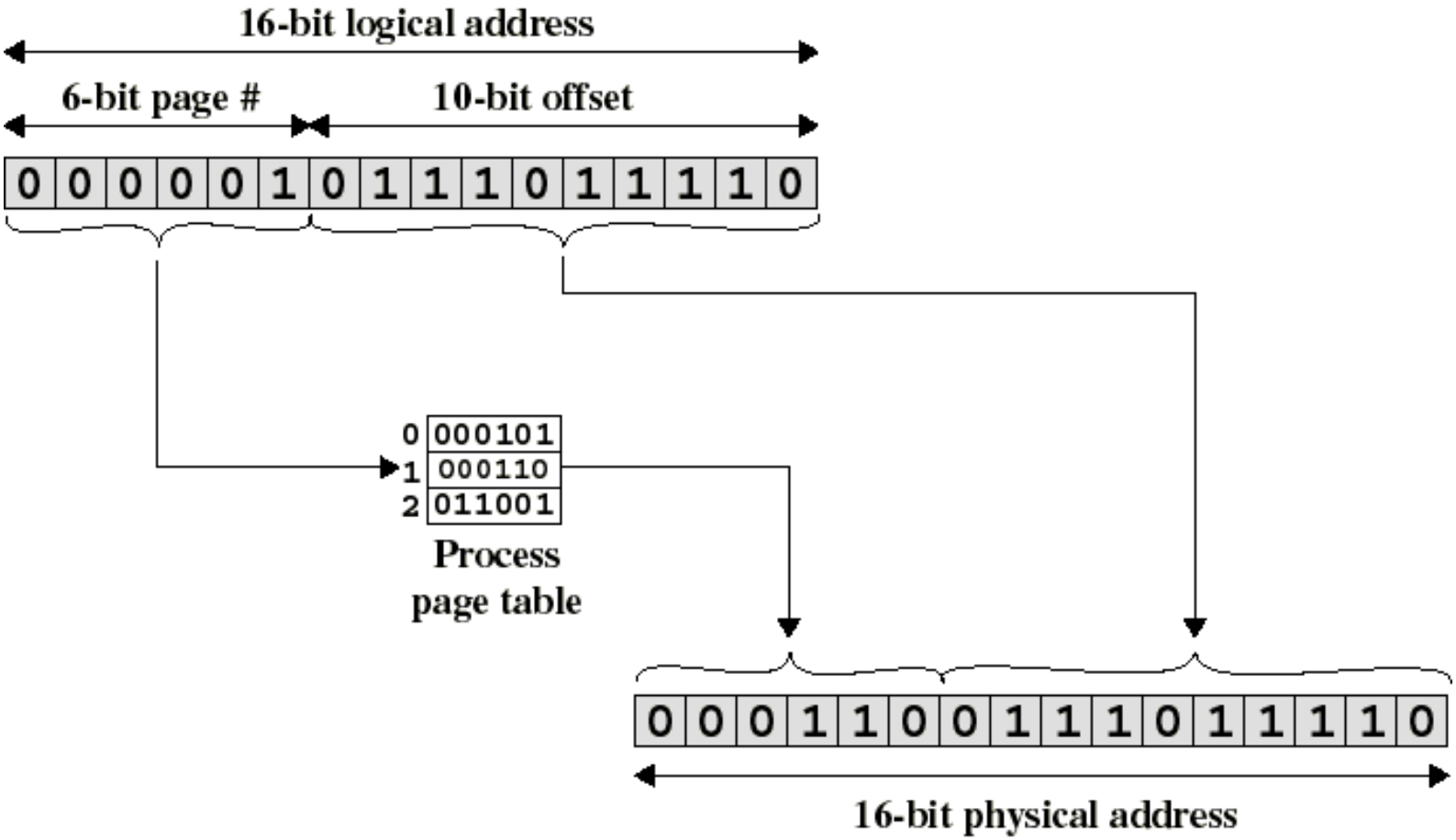
Logical address (pagination)

- **So the pages are invisible to the programmer, compiler or assembler (only relative addresses are used)**
- **Runtime address translation is easily accomplished by hardware:**
  - the logical address (n, m) is translated into a physical address (k, m) by indexing the page table and appending the same offset m to the number of the frame k
- **A program can be run on different hardware using different page sizes**
  - What changes is the interpretation of the bits by the addressing mechanism

Mechanism: hardware



Address translation (logical-physical) for pagination

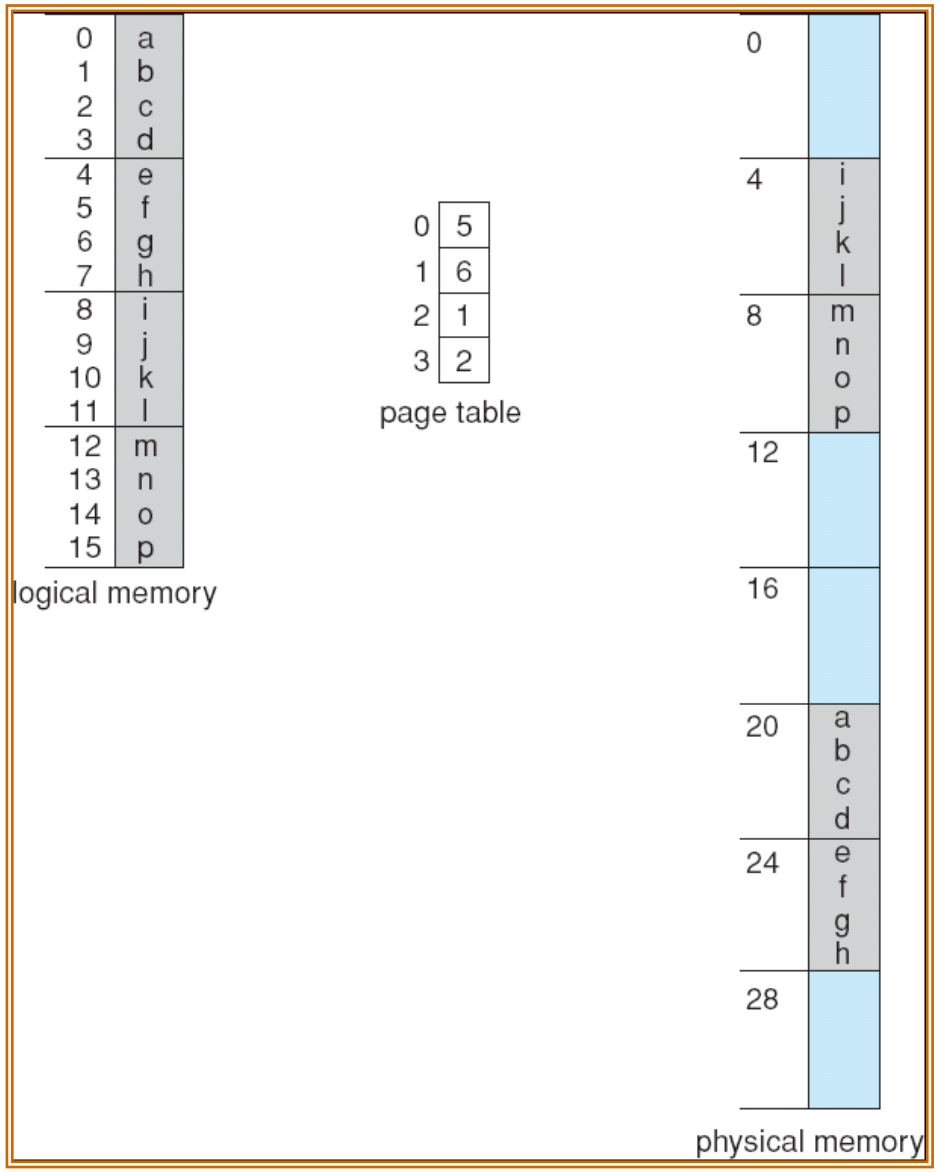


Translation of addresses: segmentation and pagination

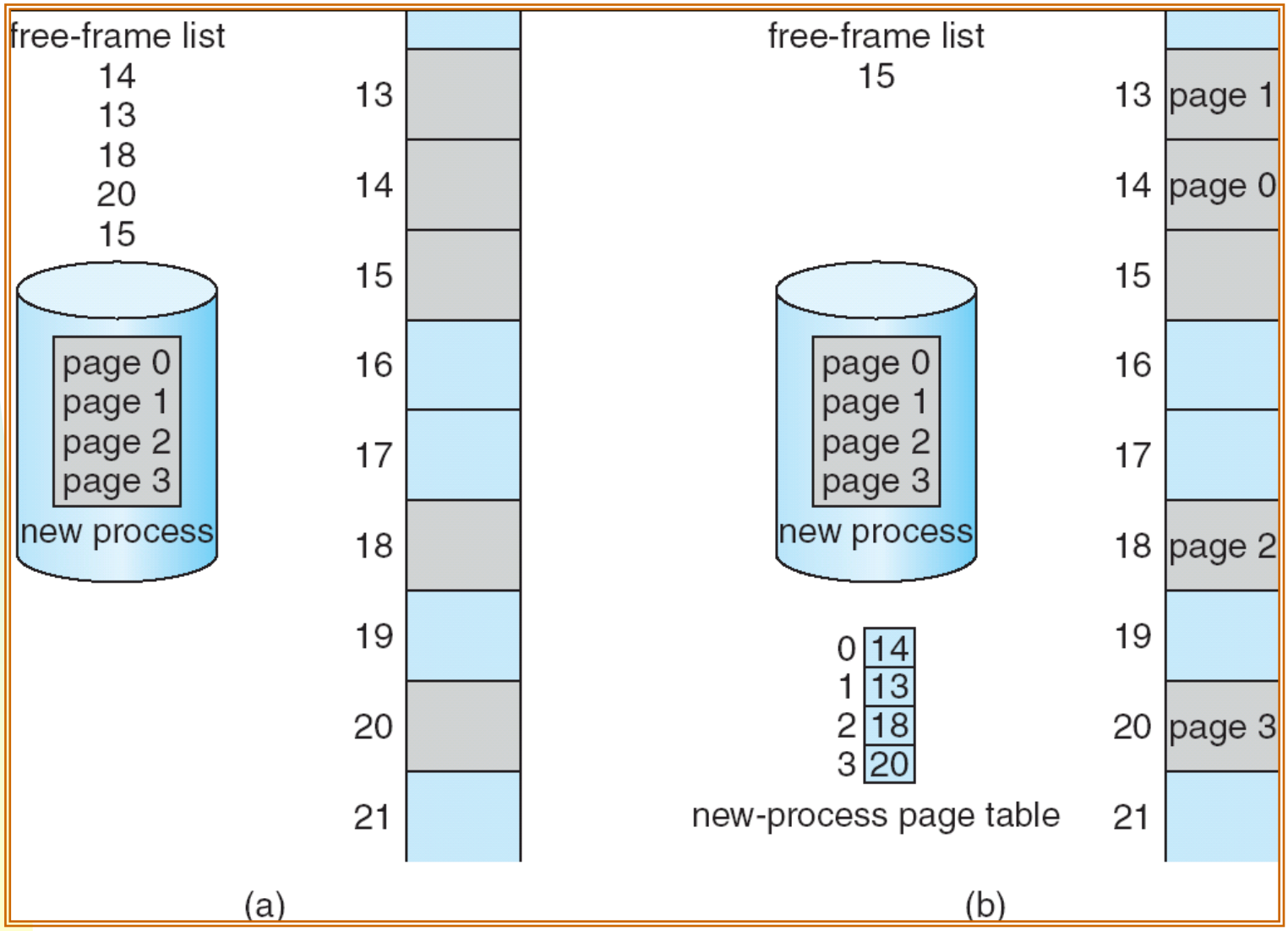
- Both in the case of segmentation and in the case of pagination, we *add* the offset to the segment or page address.
- However, in pagination, the page addresses are always multiples of 2, and there are as many 0s on the right as there are bits in the offset ... so the addition can be done by simple concatenation:

```
11010000 + 1010
=
1101 1010
```

Example of pagination



List of free frames



Efficiency issues

- **Address translation, including page and segment address lookup, is performed by hardware mechanisms**
- **However, if the page table is in main memory, each logical address causes at least 2 references to memory**
  - One to read the entry in the page table
  - The other to read the referenced word
- **The memory access time is double...**

To improve efficiency

- **Where to put the page tables (the same ideas also apply to segm tables)**
- **Solution 1: in CPU registers.**
  - advantage: speed
  - disadvantage: limited number of pages per process, memory size. logic is limited
- **Solution 2: in main memory**
  - advantage: size of the mem. unlimited logic
  - disadvantage: mentioned
- **Solution 3 (mixed): The page tables are in main memory, but the most used addresses are also in CPU registers.**

Associative registers

TLB: Translation Lookaside Buffers, or *caches* addressing

- **Parallel search for an address:**
  - the sought address is found in the left part of the table in parallel (special equipment)
- **Translation page → frame**
  - If the requested page has been used recently it will be found in the associative registers
    - Quick search

No Page	No Frame
3	15
7	19
0	17
2	23

Associatve Research in TLB

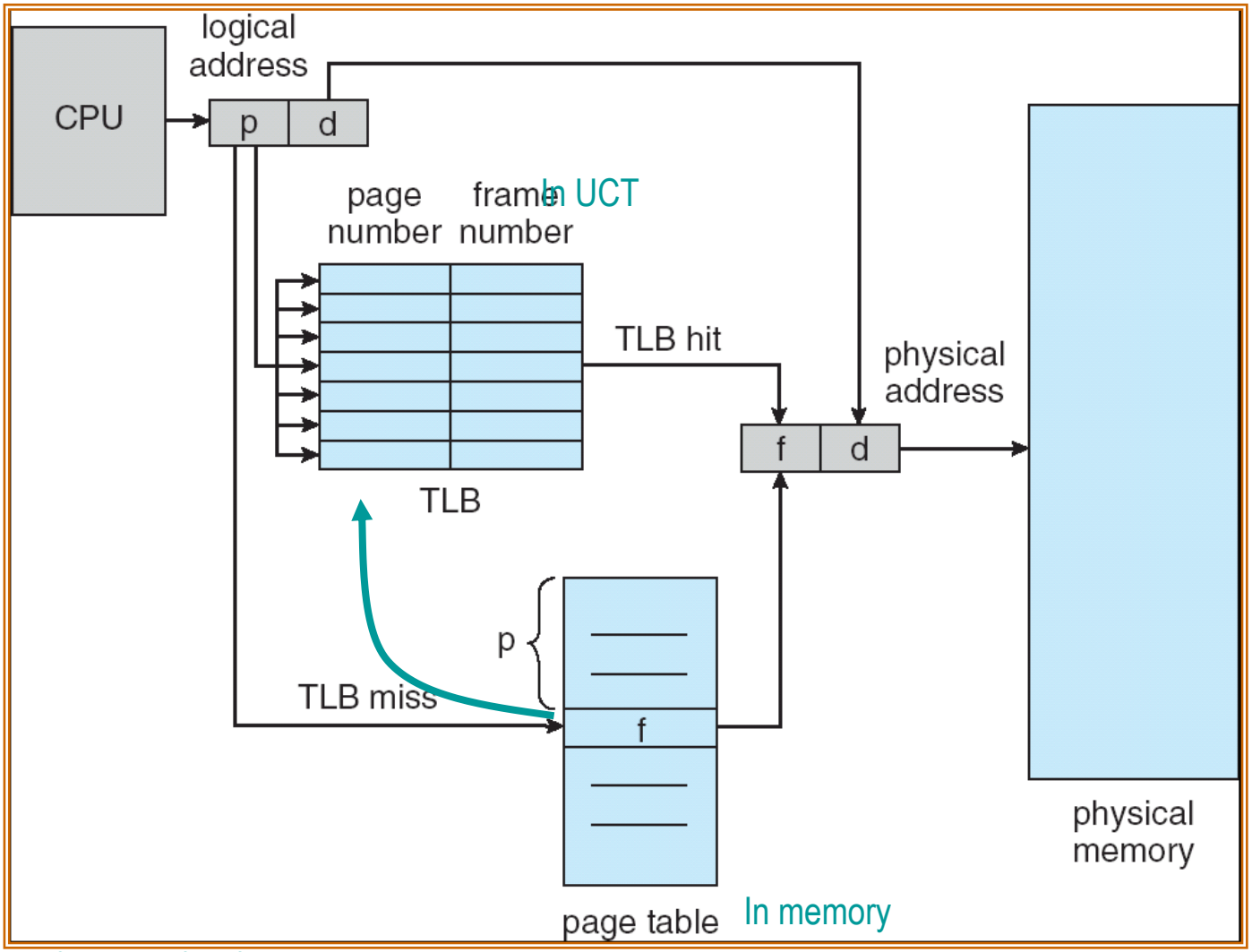
- **The TLB is a small table of hardware registers where each row contains a pair:**
  - Logical page number, Frame number
- **The TLB uses equipment from *associative memory*: simultaneous interrogation of all logical numbers to find the required physical number**
- **Each pair in the TLB is provided with a benchmark as to whether that pair has been used recently. Otherwise, it is replaced by the last pair that we need**

Translation Lookaside Buffer (TLB)

- **On receipt of a logical address, the processor examines the TLB cache**
- **If this page entry is there, the frame number is taken from it**
- **Otherwise, the page number indexes the process page table (in memory)**
  - This new page entry is put in the TLB
  - It replaces another not recently used
- **TLB is emptied when CPU changes proc**

- **The first three operations are done by material**

TLB usage diagram



In the case of 'miss', f is found in memory, then it is put in the TLB

Real access time

- **Associative search = ε time units (normally small)**
- **Suppose the memory cycle is 1 microsecond**
- **α = percentage of hits (hit ratio) = percentage of times a page number is found in associative registers**
  - this is related to the number of associative registers available

- **Effective access time t_{ea}:**

$t_{ea} = (1 + \varepsilon)\ \alpha + (2 + \varepsilon)\ (1 - \alpha)$
$= 2 + \varepsilon - \alpha$
if α is close to 1 and ε is small, this time will be close to 1.

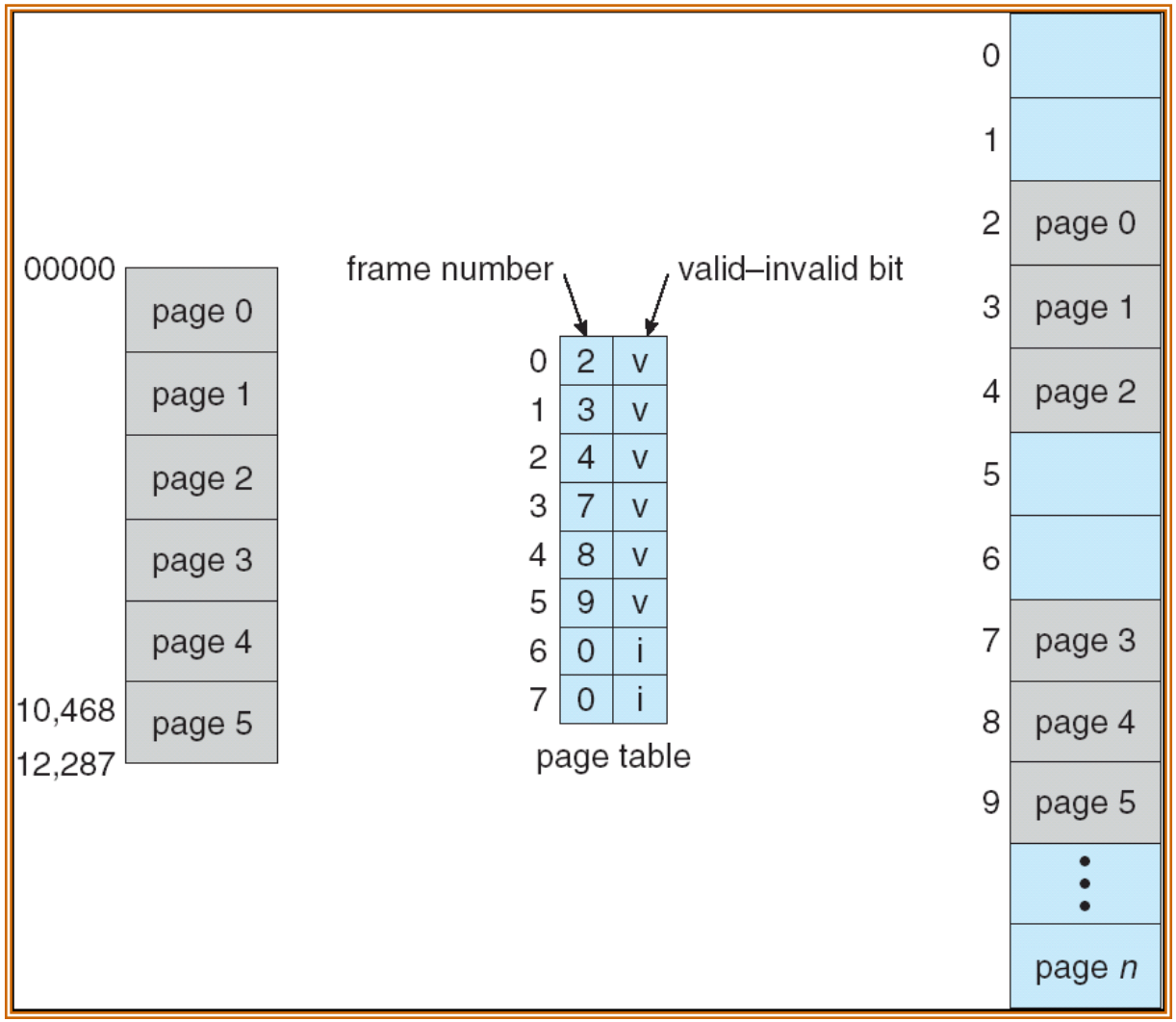
- **Generalization of the formula taking m as the access time to the main memory:**

$t_{ea} = (m + \varepsilon)\ \alpha + (2m + \varepsilon)\ (1- \alpha) = m\ \alpha + \varepsilon\alpha + 2m - 2m\ \alpha + \varepsilon - \varepsilon\alpha =$
$= 2m + \varepsilon - m\ \alpha$

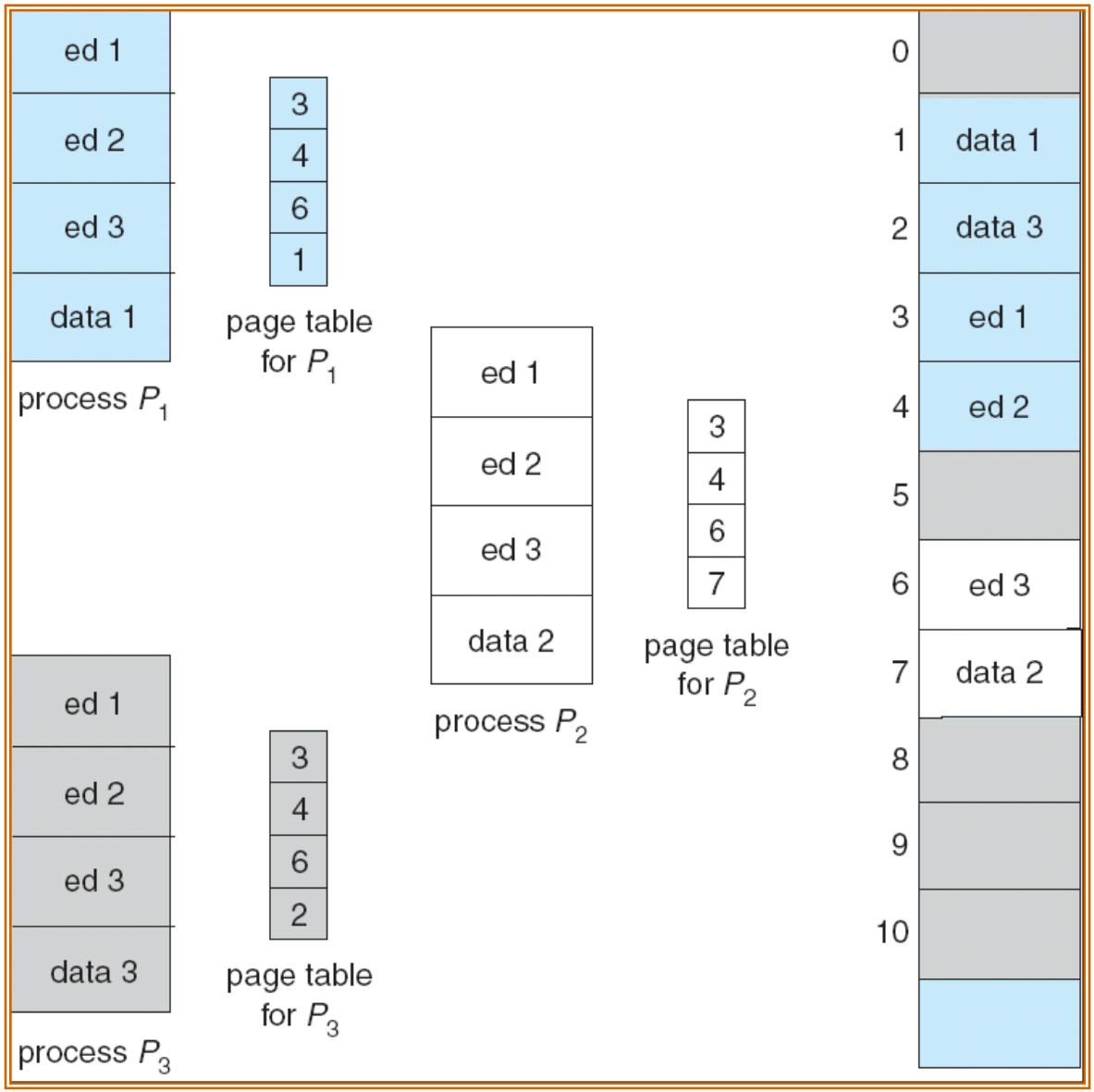
Memory protection

- **Associates various bits with each page in the page array as in the segment array**
  - E.g. valid-invalid bit indicates valid pages of the process
- **To check if an address is valid**
  - With an array of pages having a fixed size: valid-invalid bits
  - With an array of pages having a variable size (only the valid pages of the process): Compare the page # with the PTLR register (page-table length register)

Bit Valid-Invalid (vi) in the page table



Sharing pages:
3 proc. sharing an editor, on private data for each proc



Structure of page tables

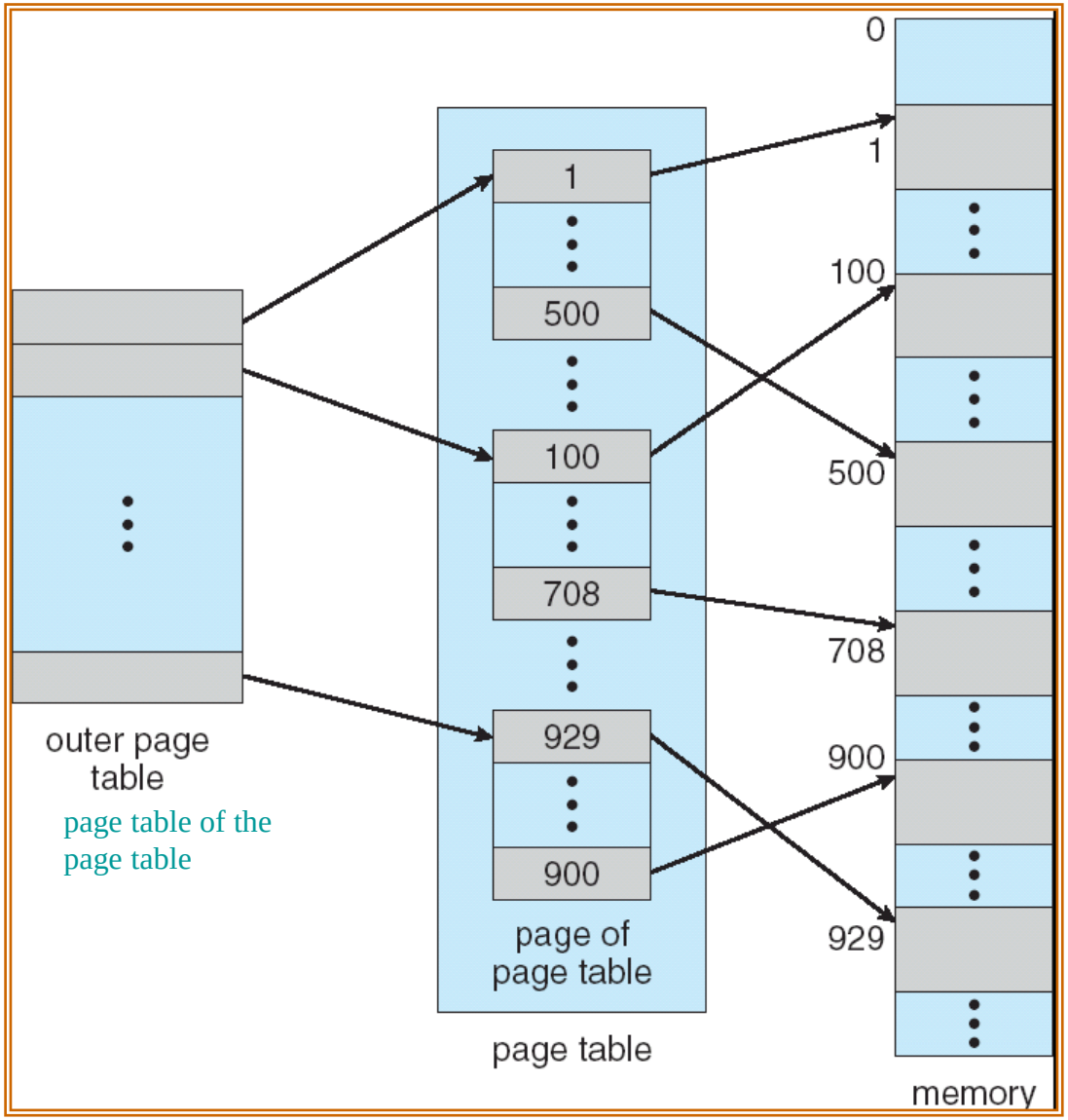
How long can an a page table be?

- 32-bit Addresses, page of 4 kbytes -> 2^{20} pages = 1M page entries!
- Oh my!

- **Hierarchical page table**

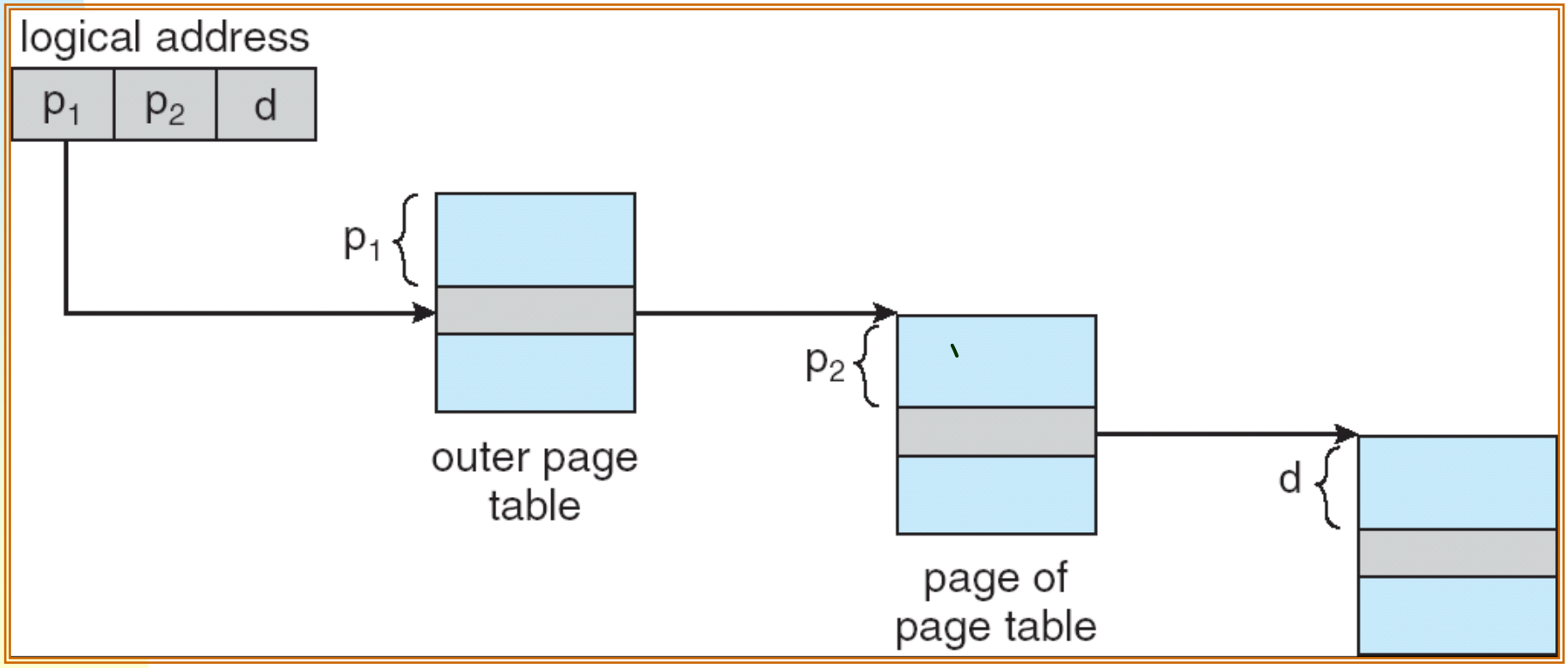
Two-level hierarchical page tables

(when the page tables are very large, they can themselves be paginated)



Hierarchical page tables on two levels

- The part of the address that belongs to the page number is itself divided into 2



Using Translation Lookaside Buffer

- In the case of multi-level paging systems, the use of TLB becomes even more important to avoid multiple memory accesses to calculate a physical address
- The most recently used addresses are found directly in the TLB.

Simple Segmentation vs Simple Pagination

- **Pagination is only concerned with the problem of loading, while**
- **Segmentation also targets the problem of connection**
- **Segmentation is visible to the programmer but pagination is not**
- **The segment is a logical unit of protection and sharing, while the page is not**
  - So protection and sharing are easier in segmentation
- **Segmentation requires more complex hardware for address translation (addition instead of concatenation)**
- **Segmentation suffers from fragmentation *external* (dynamic partitions)**
- **Pagination produces fragmentation *internal*, but not a lot (1/2 frame per program)**
- **Fortunately, segmentation and pagination can be combined**

Conclusions on Memory Management

- **Problems of:**
  - fragmentation (internal and external)
  - complexity and efficiency of algorithms
- **Two widely used methods**
  - Paging
  - Segmentation
- **Pagination and segmentation issues:**
  - size of segment tables and pages
  - pagination of these tables
  - efficiency provided by Translation Lookaside Buffer
- **The methods are often used in conjunction, resulting in complex systems**

Fragmentation recap

- **Partition fixes: internal fragmentation because partitions cannot be completely used + external fragmentation if there are unused partitions**
- **Dynamic partitions: external fragmentation which leads to the need for compression.**
- **Segmentation without pagination: no internal fragmentation, but external fragmentation due to segments of different lengths, stored contiguously (as in dynamic partitions)**
- **Pagination: on average, 1/2 frame of internal fragmentation per process**


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
Dmnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze